ERZEUGEN EINES CLUSTERS IN ORACLE

Cluster erzeugen und Clusterschlüssel angeben:

```
CREATE CLUSTER <name>(<col> <datatype>-list)
  [INDEX | HASHKEYS <integer> [HASH IS <funktion>]];
```

```
CREATE CLUSTER Cl_Sea (SeaName VARCHAR2(50));
```

- Default: *indexed Cluster*, d.h. die Zeilen werden entsprechend dem Clusterschlüsselwert indiziert und geclustert.
- Option: `HASH` mit Angabe einer Hashfunktion, nach der geclustert wird.
- Zuordnung der Tabellen mit `CREATE TABLE` unter Angabe des Clusterschlüssels:

```
CREATE TABLE <table>
  (<col> <datatype>,
    ⋮
   <col> <datatype>)
  CLUSTER <cluster>(<column-list>);
```

Erzeugen eines Clusters in ORACLE: Beispiel

```
CREATE TABLE CSea
  (Name     VARCHAR2(50) PRIMARY KEY,
   Depth    NUMBER)
  CLUSTER Cl_Sea (Name);
```

```
CREATE TABLE Cgeo_Sea
  (Province   VARCHAR2(50),
   Country    VARCHAR2(4),
   Sea        VARCHAR2(50))
  CLUSTER Cl_Sea (Sea);
```

Erzeugen des Clusterschlüsselindexes:
(Dies muss *vor* dem ersten DML-Kommando geschehen).

```
CREATE INDEX <name> ON CLUSTER <cluster>;
```

```
CREATE INDEX ClSeaInd ON CLUSTER Cl_Sea;
```

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
IS
    Declaration Section
BEGIN
    Execution Section
EXCEPTION
    Exception Section
END;
```

- 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;`
  `/`

Wichtig: nach dem Semikolon noch ein Vorwärtsslash (“/”) in einer separaten Zeile, um die Deklaration auszuführen!!!

(Beispiel → nächste Folie)

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>;
/
```

- `datatype` darf dabei nur ein atomarer SQL-Datentyp sein.
  Es können damit also keine Tabellen zurückgegeben werden.
- 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*).

PROZEDUREN UND FUNKTIONEN

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

  ```
  SHOW ERRORS;
  ```

  ausgeben lassen.

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

- 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 in sqlplus 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);
```

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(50);
```

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(50),
   Population NUMBER,
   Latitude NUMBER,
   Longitude 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.

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;
```

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 berüchtigten `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:

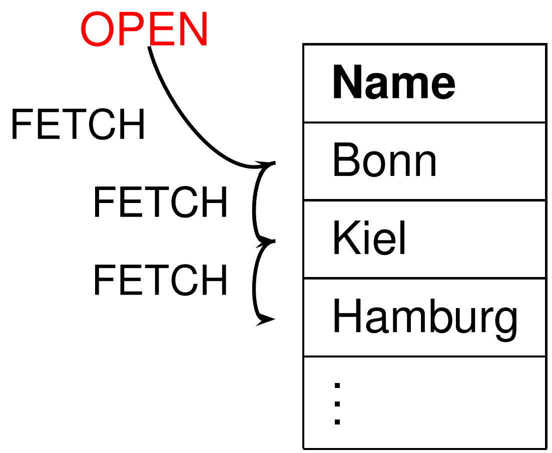
```
DECLARE
CURSOR cities_in (the_country Country.Code%TYPE)
IS SELECT Name
   FROM City
   WHERE Country=the_country;
```

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 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 geFETCHte 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).

PL/SQL-DATENTYPEN: PL/SQL TABLES

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

Typ-Definition: `TYPE <tabtype> IS TABLE OF <datatype> [INDEX BY BINARY_INTEGER];`
Var-Deklaration: `<tabname> <tabtype>;`

Built-in-Funktionen und -Prozeduren:

`<variable> := <tabname>.<built-in-function>;`
oder
`<tabname>.<built-in-procedure>;`

- COUNT (fkt): Anzahl der belegten Zeilen.
- 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.
  (ist null falls Tabelle leer ist!)
- NEXT/PRIOR(n) (fkt): Gibt ausgehend von n den nächsthöheren/nächstniedrigen belegten Indexwert.

PL/SQL Tables als einfache Collections

- dann (implizit) indiziert mit 1..n (falls nicht leer)

```
SELECT ...
BULK COLLECT INTO <tabname>
FROM ...
WHERE ...
```

```
DECLARE
 TYPE tabtype IS TABLE OF city%ROWTYPE;
 tab tabtype;
BEGIN
 SELECT *
  BULK COLLECT INTO tab
  FROM city
  WHERE country = 'D';
 IF tab.COUNT > 0 THEN
   FOR i IN tab.FIRST .. tab.LAST LOOP
     dbms_output.put_line(tab(i).name);
   END LOOP;
 END IF;
END;
/
```

[Filename: PLSQL/table1.sql]

- analog
  `TABLE OF <table>.<attr>%TYPE`
  und dann Zugriff nur mit `tab(i)`.
- bei BULK COLLECT wird der vorherige Inhalt der Tabelle überschrieben.

PL/SQL Tables als indizierte Collections

```
TYPE <tabtype> IS TABLE OF <datatype>
  INDEX BY BINARY_INTEGER;
<tabname> <tabtype>;
```

- Adressierung: `<tabname>`(n)
- *sparse*: nur die Zeilen gespeichert, die Werte enthalten.
- Dann springen mit `WHILE` und `<tabname>.next`:

```
DECLARE
 TYPE plz_table_type IS TABLE OF City.Name%TYPE
      INDEX BY BINARY_INTEGER;
 plztab plz_table_type;
 i NUMBER;
BEGIN
  plztab(37077):= 'Goettingen';
  plztab(79110):= 'Freiburg';
  plztab(33334):= 'Kassel';
  i := plztab.first;              -- 33334
 WHILE NOT i IS NULL LOOP
  dbms_output.put_line(i || ' ' || plztab(i));
  i := plztab.next(i);
 END LOOP;
END;
/
```

[Filename: PLSQL/table2.sql]

PL/SQL Tables

- Tabellen können auch als Ganzes zugewiesen werden

```
andere_table := plz_table;
```

- Unterschied BULK COLLECT zu Cursor:
  - Cursor wird on-demand iteratorbasiert ausgewertet, kann abgebrochen werden,
  - BULK COLLECT wertet komplett aus und legt das Ergebnis in PL/SQL-Tabelle ab.
    - BULK COLLECT ist daher ineffizienter, gibt aber die DB sofort wieder frei (falls eine andere Transaktion schreibend zugreifen möchte).

PL/SQL Tables als Rückgabewert einer Funktion

```
CREATE OR REPLACE TYPE membership_type AS OBJECT (
       country VARCHAR2(4),
       type VARCHAR2(60) );
/
CREATE OR REPLACE TYPE memberships_type AS TABLE OF membership_type;
/
CREATE OR REPLACE FUNCTION members_of(org VARCHAR2)
RETURN memberships_type
IS
  toreturn memberships_type;
BEGIN
  SELECT membership_type(country, type)
    BULK COLLECT INTO toreturn
    FROM ismember
    WHERE organization = org;
  RETURN toreturn;
END;
/
SELECT country, type
FROM TABLE(SELECT members_of('EU') FROM DUAL);
```

[Filename: PLSQL/return_table.sql]

8.3 Dynamic SQL

- `EXECUTE IMMEDIATE <string>`
- ‹string› kann dabei eine Konstante sein,
  - DDL-Statements werden in PL/SQL nicht direkt unterstützt:

```
BEGIN
 execute immediate 'drop table continent';
END;
```

- oder kann dynamisch zusammengesetzt werden (Bsp. nächste Folie)
- und Platzhalter für Werte, die dann bei Ausführung eingesetzt werden, enthalten

```
DECLARE country VARCHAR2(4) := 'CDN';
        org VARCHAR2(10) := 'EU';
BEGIN
 execute immediate
     'insert into isMember VALUES (:1, :2, :3)'
   using country, org, 'candidate';
END;
/
```

Dynamic SQL: Kommandos als String erstellen und ausführen

```
CREATE OR REPLACE PROCEDURE clean
IS
 BEGIN
  FOR tn IN
   ( SELECT table_name FROM all_tables
     WHERE table_name LIKE 'TMP_%')
  LOOP
   execute immediate 'DROP TABLE ' || tn.table_name;
  END LOOP;
END;
/
```

[Filename: PLSQL/clean.sql]

Dynamic SQL: Beispiel

- Wert in eine PL/SQL-Variable einlesen: INTO

```
CREATE OR REPLACE PROCEDURE getpop (cname City.name%TYPE)
IS
  cty city%ROWTYPE;
  BEGIN
  execute immediate 'select * from city where name= :1'
    into cty
    using cname;
  dbms_output.put_line(cty.population);
  END;
/
execute getpop('Berlin');
```

[Filename: PLSQL/dynamicselect.sql]

8.4 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.

- Entsprechende Privilegien muss man direkt (GRANT ... TO ‹user›), und nicht nur über eine Rolle bekommen haben.

8.5 Geschachtelte Tabellen unter PL/SQL

Nested_Spoken		
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_Spoken* 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_Spoken
               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.

8.6 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.

Trigger

```
CREATE [OR REPLACE] TRIGGER <trigger-name>
  BEFORE | AFTER
  {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.

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 DURCH EXCEPTIONS IN PL/SQL

- 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 SYSDATE
     BETWEEN to_date('12:00', 'HH24:MI')
         AND to_date('18:00', 'HH24:MI')
  THEN RAISE_APPLICATION_ERROR
         (-20101,'Unerlaubte Aktion');
  END IF;
END;
/
```

[Filename: PLSQL/trigger-nachmittag.sql]

Beispiel

```
CREATE OR REPLACE TRIGGER dummytrigger
INSTEAD OF INSERT ON AllCountry
FOR EACH ROW
BEGIN
 -- username must be in capitals!!!
 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)

8.7 Zeitgesteuerte Jobs

(eigentlich nicht direkt zu PL/SQL gehörend)

- Implementierung zeitabhängiger Trigger,
- bei jedem Produkt anders.
- Beschreibung hier für Oracle (Stand 12c, 2014)

Jobs und Schedules in Oracle

- Man benötigt die Systemprivilegien CREATE JOB, MANAGE SCHEDULER, ggf. auch CREATE EXTERNAL JOB
- die Konfiguration benutzt eine objektorientierte Syntax (ist auch auf Basis der objektrelationalen Interna umgesetzt)
- interne (PL/SQL) und externe (Aufruf eines Programmes via Pfad) Jobs möglich
- komplexe Spezifikationen von Aufrufzeiten via Schedules möglich.

Einmalige zeitgesteuerte Jobs: Beispiel

- der folgende Job trägt nach einer Minute die (dann) aktuelle Zeit in die Tabelle “jobtest” ein, danach wird der Job gelöscht.
- Syntax in SQLplus:

  `execute DBMS_SCHEDULER.CREATE_JOB (<job-spezifikation>);`

  ... dabei werden aber keine Zeilenumbrüche akzeptiert.
- also dasselbe (didaktisch) besser in einen PL/SQL-begin/end-Block packen:

```
create table jobtest (x DATE);
begin
  DBMS_SCHEDULER.CREATE_JOB
   (job_name => 'job1',
    job_type => 'PLSQL_BLOCK',
    job_action => 'begin insert into jobtest
                       values (SYSDATE); end;',
    start_date => SYSDATE+1/1440,
    enabled => TRUE);
end;
/
```

[Filename: PLSQL/simple-job.sql]

Job-Attribute und Aufruf via Scheduler

- enabled: TRUE aktiviert sofort, FALSE hält einen Job deaktiviert

  ```
  execute DBMS_SCHEDULER.ENABLE('job1');
  execute DBMS_SCHEDULER.DISABLE('job1');
  ```

  manuell aufrufen bzw löschen:

  ```
  execute DBMS_SCHEDULER.RUN_JOB('job1');
  execute DBMS_SCHEDULER.DROP_JOB('job1');
  ```

  Fehlermeldungen erhält man nur bei manuellem Aufruf!

- start_date, end_date: ggf. Anfang und Ende.
- auto_drop: default TRUE; FALSE sorgt für Wiederholung.
- repeat_interval: basierend auf "Schedules"; Details siehe Dokumentation, z.B.
  - FREQ = YEARLY|MONTHLY|...|SECONDLY;   Basisangabe, wie oft,
  - INTERVAL = 1..99   jedes, jedes zweite, jedes 99. von FREQ,
  - BYMONTH =...; BYDAY=...;   Spezifikation des "wann" innerhalb des "wie oft",

  `repeat_interval => 'FREQ = WEEKLY; INTERVAL = 2;`   jede zweite Woche
  `BYDAY = MON, THU; BYHOUR = 15, 16; BYMINUTE = 00'`   Montags+Donnerstags, 15+16 Uhr

- job_type: 'PL/SQL_BLOCK', 'STORED_PROCEDURE', 'EXECUTABLE' (externer Job)

Wiederholende zeitgesteuerte Jobs: Beispiel

- der folgende Job erhöht alle zwei Minuten zu jeweils 5 angegebenen Sekundenzeitpunkten die Bevölkerung um 1:

```
begin
 DBMS_SCHEDULER.DROP_JOB('job2');
 DBMS_SCHEDULER.CREATE_JOB
  (job_name => 'job2',
   job_type => 'PLSQL_BLOCK',
   job_action => 'begin
                  update country set population = population + 1 where code=''CN'';
                  end;',
   auto_drop => FALSE,
   repeat_interval => 'FREQ = MINUTELY; INTERVAL = 2;
     BYSECOND = 5, 18, 31, 45, 51',
   start_date => SYSDATE+1/2880,  -- after 30 secs
   end_date => SYSDATE+11/1440, -- after 11 minutes
   enabled => TRUE);
end;
/
```

[Filename: PLSQL/repeating-job.sql]

- ... in der 11. Minute nur noch 2x. Zusammen also 5x5 (0,2,4,6,8 min) + 1x2 = 27x.

Externe Jobs

- siehe Dokumentation.
- benötigt das Recht CREATE EXTERNAL JOB,
- werden auf dem Rechner ausgeführt, auf dem Oracle läuft,
- Oracle führt einen Benutzerwechsel durch (konfigurierbar, default: nobody/nogroup),
- job_action enthält nur Pfad des auszuführenden Programms, Parameter müssen separat übergeben werden,
- Fehlermeldungen erhält man nur bei manuellem Aufruf mit

  ```
  execute DBMS_SCHEDULER.RUN_JOB('job1');
  ```

Informationen über bestehende Jobs

```
SELECT job_name, job_action
FROM user_scheduler_jobs;
```

8.8 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.

8.9 Scripting: SQLplus und PL/SQL

- SQLPlus ist Oracle's *Client* für SQL,
  - Interface für Queries,
  - Ausführen von Skripten aus dem Filesystem des *Benutzers*.
    (z.B. create.sql)
- PL/SQL ist die *server-interne* Skriptsprache.
  - *innerhalb SQL aufrufbare Prozeduren/Funktionen/Trigger*
  - anonyme BEGIN ... END-Blöcke
- Man kann beide (ein bisschen) kombinieren.
  PL/SQL wird dann client-seitig ausgeführt.

PL/SQL in SQLPlus-Skripten

- PL/SQL `begin ... end;`-Blöcke können in SQLPlus-Skripten enthalten sein, und werden auf dem *Client* ausgewertet.
  Um ausgeführt zu werden, müssen sie mit "`/`" abgeschlossen werden.

Aufruf von Skripten (im *lokalen Filesystem*) aus PL/SQL in SQLPlus-Skripten

- Mit `@` (wie in SQLplus; wobei `start` hier nicht erlaubt ist!),
- Das aufgerufene Skript muss dann PL/SQL-konform sein
  - einige SQL-Kommandos, die in SQLplus direkt zulässig sind, jedoch nicht in PL/SQL, müssen in
    `execute immediate '...';`
    eingeschlossen sein.

Beispiel: Das create.sql-Skript

```
spool create.log
-- note: the file is located in the directory where the
-- *files* are located,
-- i.e. /afs/informatik.uni-goettingen.de/group/dbis/public/Mondial/create.log
start mondial-drop-tables;
start mondial-schema;
start mondial-inputs;
set serveroutput on;
 -- set read-all for DBIS user
BEGIN
 IF user = 'DBIS'  -- capitalized!
   THEN
    dbms_output.put_line('dbis: set grants');
    @mondial-grants.plsql;
 ELSE dbms_output.put_line('not dbis, no grants');
 END IF;
END ;
/
spool off
prompt Logfile 'create.log' created ...
```

- `$ORACLE_PATH` muss das Directory enthalten, in dem die o.g. Dateien liegen
- mondial-grants.plsql ist ein PL/SQL-Fragment

[Filename: SQLPlus/create.sql]

Beispiel (cont'd)

- mondial-grants.sql ist ein SQLplus-Skript, das einfach nur das PL/SQL-Skript aufruft:

```
BEGIN
  @mondial-grants.plsql;
END ;
/
```

[Filename: SQLPlus/mondial-grants.sql]

- mondial-grants.plsql ist ein PL/SQL-Skript, das von anderen Skripten (innerhalb eines PL/SQL-begin-end-Blocks) aufgerufen werden kann.
  (es enthält *kein* "/" am Ende!)
  (in einem reinen SQLPlus-Skript könnte "GRANT ..." alleine stehen)

[Filename: SQLPlus/mondial-grants.plsql]

```
execute immediate 'GRANT EXECUTE ON geocoord TO student';

execute immediate 'GRANT SELECT ON Country TO student';
execute immediate 'GRANT SELECT ON City TO student';
execute immediate 'GRANT SELECT ON Province TO student';
execute immediate 'GRANT SELECT ON Economy TO student';
execute immediate 'GRANT SELECT ON Population TO student';
execute immediate 'GRANT SELECT ON Politics TO student';
execute immediate 'GRANT SELECT ON Language TO student';
```

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/8.1.5: auch Java, siehe Folie 337)
    - `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 409).

STUFEN DER OBJEKTORIENTIERUNG

"Konservative" objektrelationale Erweiterungen (seit Oracle 8)

(siehe Folie 256)

- 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 279)

- 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,

Typ-Implementierung

- **Type Body:** Implementierung der Methoden in PL/SQL
- seit Oracle 8i auch in PL/SQL+Java (siehe Folien 337 und 343)

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>]
  );
/
```

← *dieser Slash ist unbedingt notwendig!*

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

Beispiel: Geo-Koordinaten

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

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

[Filename: ObjRel/geocoord-type.sql]

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

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

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

- `CASCADE`: “forwards” definition to tables and data objects where it is used.

TYPE BODY

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

Funktionen: dürfen den Datenbankzustand nicht verändern,

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

$\Rightarrow$ verwenden nur den Zustand der beteiligten Objekte.

Type Body

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

Beispiel: Geo-Koordinaten

```
CREATE OR REPLACE TYPE BODY GeoCoord
AS
MEMBER FUNCTION Distance (other IN GeoCoord) RETURN NUMBER
  IS
  BEGIN
   RETURN 6370 * ACOS(COS(SELF.latitude/180*3.14)
               * COS(other.latitude/180*3.14)
               * COS((SELF.longitude - other.longitude)/180*3.14)
               + SIN(SELF.latitude/180*3.14) * SIN(other.latitude/180*3.14));
  END;
MAP MEMBER FUNCTION Distance_Greenwich RETURN NUMBER
  IS
  BEGIN
   RETURN SELF.Distance(GeoCoord(51.5, 0));
  END;
END;
/
```

[Filename: ObjRel/geocoord-body.sql]

ERZEUGUNG VON OBJEKTEN

- Konstruktormethode: `<type>(<arg_1>, ..., <arg_n>)`

  Also kein `NEW`, sondern nur einfach

```
GeoCoord(48,8)
CityORType('Berlin', 'Berlin', 'D', 3472009, GeoCoord(52.45, 13.3))
```

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(52.45,-30).Distance(geoCoord(52.45,-31))
FROM DUAL;
SELECT geoCoord(0,-30).Distance(geoCoord(0,-31))
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,
    Elevation NUMBER,
    Coordinates GeoCoord CONSTRAINT MountainCoord
       CHECK ((Coordinates.Latitude >= -90) AND
              (Coordinates.Latitude <= 90) AND
              (Coordinates.Longitude > -180) AND
              (Coordinates.Longitude <= 180)));
```

[Filename: ObjRel/mountain-table.sql]

- Constraints werden wie immer bei der Tabellendefinition angegeben.

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

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

ZEILENOBJEKTE

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

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

mit `<constraint-list>` wie bisher:

- attributbezogene Bedingungen entsprechen den Spaltenbedingungen:

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

- Tabellenbedingungen: Syntax wie bei Tupeltabellen.

Zeilenobjekte

Beispiel: *CityORType*

Objekt-Relationaler City-Typ:

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

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

[Filename: ObjRel/cityORtype.sql]

Zeilenobjekte

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

[Filename: ObjRel/cityORtypebody.sql]

OBJEKTTABELLEN: ZEILENOBJEKTE

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

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

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

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

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

Verwendung von Objekttabellen

Auslesen und Ändern von Attributwerten wie bekannt

- Auslesen:

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

- Ändern:

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

Methodenaufrufe wie erwartet

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

Verwendung von Objekttabellen

... Auslesen von Objekten als Objekte:

Das folgende geht so *nicht*:

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

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

Die VALUE()-Funktion

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

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

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

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

Verwendung von Objekten: VALUE

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

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

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

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

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

9.1.3 `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.
- Für jeden Objekttyp eine `MAP FUNCTION` oder eine `ORDER FUNCTION`.

`MAP`**-Funktion:** (Betragsfunktion)

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

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

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

- `MAP`- und `ORDER`-Funktionen dürfen *keinen Datenbankzugriff* enthalten.

MAP-Methoden: Beispiel

MAP-Methode auf *GeoCoord*:

```
CREATE OR REPLACE TYPE BODY GeoCoord
AS
 :
MAP MEMBER FUNCTION Distance_Greenwich RETURN NUMBER
  IS
  BEGIN
   RETURN SELF.Distance(GeoCoord(51.5, 0));
  END;
END;
/
```

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

[Filename: ObjRel/orderby.sql]

ORDER-METHODEN

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

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

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

VERGLEICHE ZWISCHEN OBJEKTEN

- $<$, $>$, “between” und Ordnung basieren auf der MAP bzw. ORDER-Methode.
- Gleichheit/Ungleichheit:
  - bei Zeilenobjekten: diese haben eine Objektidentität; Vergleich mit VALUE(obj_1) = VALUE(obj_2).
  - Anmerkung: bei Spaltenobjekten, wie z.B. Instanzen von GeoCoord, wäre ein Vergleich auf Objektidentität nicht sinnvoll.
  - der Vergleich $obj_1 = obj_2$ basiert auf der MAP bzw. ORDER-Methode.

⇒ Fehler beim Compilieren, falls keine solche definiert ist!

⇒ falls eine solche definiert ist, wird sie verwendet, und damit “betragsgleiche” Objekte als gleich behandelt!

⇒ “Echten” Vergleich ggf. ausprogrammieren auf Basis der “Schlüsselattribute” der Objekte (ggf. in equals(...)-Methode).

9.1.4 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 290 analog ausprogrammiert)

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

  organization.isMember($carcode$)
  country.isMemberIn(org-$abbrev$)
  organization.makeMember($carcode$)
  country.makeMemberIn(org-$abbrev$)

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

  Hinweis: Boolesche Anfragen der Art “Ist x Mitglied in y” sind damit möglich. Es ist jedoch keine Methode “alle Mitglieder von y möglich – diese müsste eine Relation bzw. Menge zurückgeben.
- ... man kann aber diese Implementierung dann auch beliebig ändern.

9.2 Objektorientierte Modellierung

bisher: 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.
- Erzeugen einer Referenz (Selektieren einer OID):

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

OBJEKTREFERENZEN: CONSTRAINTS

- Objekttyp kann in verschiedenen Tabellen vorkommen:
  Einschränkung auf eine bestimmte Tabelle bei der Deklaration der entsprechenden Tabelle als Spalten- oder Tabellenconstraints mit `SCOPE`:
  - als Spaltenconstraint (nur bei Tupeltabellen):
    ```
    <ref-attr> REF <object-datatype>
      SCOPE IS <object-table>
    ```
  - als Tabellenconstraint:
    ```
    SCOPE FOR (<ref-attr>) IS <object-table>
    ```
- problemlose Integration referentieller Integritätsbedingungen von Objekttabellen zu bestehenden relationalen Tabellen.
- `PRIMARY KEY`s dürfen keine `REF`-Attribute umfassen.
- Referentielle Integritätsbedingung
  ```
  FOREIGN KEY (<ref-attr>) REFERENCES <object-table>
  ```
  geht auch, wenn auf `<object-table>` kein Primary Key definiert ist (verwendet Object-ID).

Beispiel: Objekttyp *Organization*

```
CREATE TYPE Member_Type AS OBJECT
 (Country VARCHAR2(4),
  Type VARCHAR2(60));
/
CREATE TYPE Member_List_Type AS
 TABLE OF Member_Type;
/
CREATE OR REPLACE TYPE Organization_Type AS OBJECT
  ( Name VARCHAR2(80),
    Abbrev VARCHAR2(12),
    Members Member_List_Type,
    Established DATE,
    hasHqIn REF CityORType,
    MEMBER FUNCTION isMember (the_country IN VARCHAR2) RETURN VARCHAR2,
    -- EU.isMember('SLO') = 'membership applicant'
    MEMBER FUNCTION people RETURN NUMBER,
    MEMBER FUNCTION numberOfMembers RETURN NUMBER,
    MEMBER PROCEDURE addMember (the_country IN VARCHAR2, the_type IN VARCHAR2));
/
```

[Filename: Obj/org-type.sql]

Beispiel: Objekttyp *Organization*

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

- Type Body noch nicht definiert. 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 *hasHqIn*:

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

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

SELEKTION VON OBJEKTATTRIBUTEN

- Wertattribute

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

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

- Referenzattribute:

  `SELECT <ref-attr-name>`

  liefert OID:

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

Name	Abbrev	hasHqIn
European Community	EU	`<oid>`

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

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

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

VERWENDUNG VON REFERENZATTRIBUTEN

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

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

  adressiert ("*navigierender Zugriff*").

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

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

Abbrev	**hasHqIn.Name**
EU	Bruxelles

- Auch erlaubt:

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

Die Funktionen VALUE, REF, DEREF

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

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

und

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

ZYKLISCHE REFERENZEN

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

- `City_Type: country REF Country_Type`
- `Country_Type: capital REF City_Type`
- Deklaration jedes Datentypen benötigt bereits die Definition des anderen.
- Definition von *unvollständigen* Typen
  "Forward-Deklaration":

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

- wird später durch eine komplette Typdeklaration ergänzt.

Unvollständige Datentypen

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

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

- Die Nutzung in Referenzen ist damit erlaubt:

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

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

- Die *direkte Nutzung* wäre erst erlaubt, wenn City_type komplett ist:

```
CREATE TYPE city_list_2 AS TABLE OF City_type;
/ - waere eine Tabelle von City-Objekten
```

```
CREATE OR REPLACE TYPE Country_Type_2 AS OBJECT
  ( Name VARCHAR2(50),
    Code VARCHAR2(4),
    Capital City_Type);
/ - Capital waere ein Spaltenobjekt
```

Zyklische Referenzen: Beispiel

```
CREATE OR REPLACE TYPE City_Type
/

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

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

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

OBJEKTORIENTIERUNG: MODELLIERUNGSASPEKTE

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

Beispiel/Aufgabe

- City, Country, Organization als Objektgraph
- Beziehungen immer über Methoden behandeln:

  organization.isMember($carcode$)
  country.isMemberIn(org-$abbrev$)
  organization.makeMember($carcode$)
  country.makeMemberIn(org-$abbrev$)

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

9.3 Methoden: Funktionen und Prozeduren

`TYPE BODY` enthält die Implementierungen der Methoden in PL/SQL

Anpassung von PL/SQL an Objektrelationale Features

- PL/SQL unterstützt keine Navigation entlang Pfadausdrücken (in SQL ist es erlaubt).
- Jede `MEMBER METHOD` besitzt einen *impliziten* Parameter `SELF`, der das jeweilige Host-Objekt referenziert.
- Tabellenwertige Attribute können innerhalb PL/SQL wie PL/SQL-Tabellen behandelt werden:

  Built-in Methoden für Collections (PL/SQL-Tabellen) können auch auf tabellenwertige Attribute angewendet werden:

  `<attr-name>.COUNT`: Anzahl der in der geschachtelten Tabelle enthaltenen Tupel

  Verwendung in in PL/SQL eingebetteten SQL-Statements – z.B. `SELECT <attr>.COUNT` – nicht erlaubt.
- Weitere Erweiterung: Java (siehe Folie 337).

Member-Methods: Beispiel

```
CREATE OR REPLACE TYPE BODY Organization_Type IS

MEMBER FUNCTION isMember (the_country IN VARCHAR2)
  RETURN VARCHAR2
IS
BEGIN
  IF SELF.Members IS NULL OR SELF.Members.COUNT = 0
    THEN RETURN 'no'; END IF;
  FOR i in 1 .. Members.COUNT
  LOOP
    IF the_country = Members(i).country
       THEN RETURN Members(i).type; END IF;
  END LOOP;
  RETURN 'no';
END;                                    (bitte umblättern)
```

Member-Methods: Beispiel (Forts.)

```
MEMBER FUNCTION people RETURN NUMBER IS
p NUMBER;
BEGIN
  SELECT SUM(population) INTO p
  FROM Country ctry
  WHERE ctry.Code IN
  ( SELECT Country
    FROM THE (SELECT Members
                 FROM Organization_ObjTab org
                 WHERE org.Abbrev = SELF.Abbrev));
  RETURN p;
END;

MEMBER FUNCTION numberOfMembers RETURN NUMBER
IS
BEGIN
  IF SELF.Members IS NULL THEN RETURN 0; END IF;
  RETURN Members.COUNT;
END;                                    (bitte umblättern)
```

Member-Methods: Beispiel (Forts.)

```
MEMBER PROCEDURE addMember (the_country IN VARCHAR2, the_type IN VARCHAR2) IS
BEGIN
  IF NOT SELF.isMember(the_country) = 'no'
    THEN RETURN; END IF;
  IF SELF.Members IS NULL THEN
    UPDATE Organization_ObjTab
    SET Members = Member_List_Type()
    WHERE Abbrev = SELF.Abbrev;
  END IF;
  INSERT INTO
  THE (SELECT Members
    FROM Organization_ObjTab org
    WHERE org.Abbrev = SELF.Abbrev)
  VALUES (the_country, the_type);
END;
END;
/
```

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

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

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

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

Prozeduren

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

⇒ freie Prozeduren in PL/SQL als *Wrapper*, um `MEMBER PROCEDURES` aufzurufen

Beispiel: Freie Prozedur als Wrapper eines Objektmethodenaufruf

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

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

[Filename: Obj/makemember.sql]

Beispiel: Füllen der Objekttabelle

Übertragung des Datenbestandes aus den relationalen Tabellen *Organization* und *isMember* 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 mship IN (SELECT * FROM isMember)
 LOOP makeMember(mship.organization, mship.country, mship.type);
 END LOOP;
END;
/
EXECUTE Insert_All_Members;
UPDATE Organization_ObjTab org
SET hasHqIn =
   ( SELECT REF(cty)
     FROM ORCity cty, Organization old
     WHERE org.Abbrev = old.Abbreviation
     AND cty.Name = old.City AND cty.Province = old.Province AND cty.Country = old.Country);
```

[Filename: Obj/fill-organizations.sql]

Beispiel: Nutzung freier Methoden

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

[Filename: Obj/is-member.sql]

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

isMemberIn('EU', 'TR')
candidate

Es ist (zumindest bis ORACLE 19 – (20.3.2022)) nicht möglich, durch Navigation mit Pfadausdrücken Tabelleninhalte zu verändern:

```
UPDATE Organization_ObjTab org
SET org.hasHqIn.Name = 'Big Apple'  -- NICHT ERLAUBT
WHERE org.Abbrev = 'UN';
```

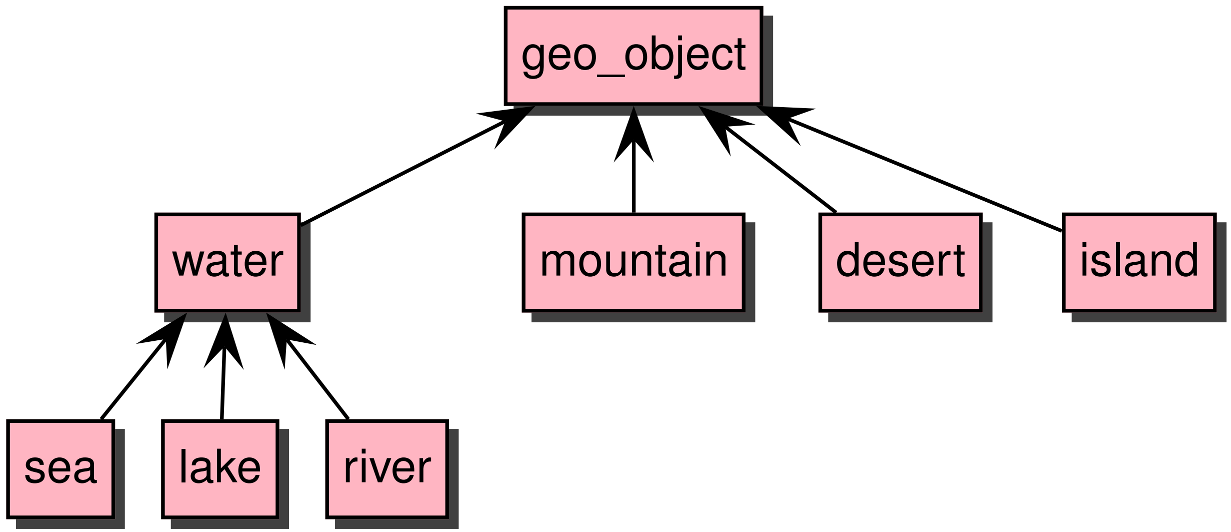
MODELLIERUNG VS. IMPLEMENTIERUNG

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

⇒ für den Benutzer bleiben die freien Methoden dieselben.

9.4 Klassenhierarchie und Vererbung

(Dieser Abschnitt ist optional)



KLASSENHIERARCHIE UND VERERBUNG

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

Erzeugen von Subtypen

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

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

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

Erzeugen von Subtypen

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

- + Angabe eines TYPE BODY, der `bla` implementiert.

Erzeugen von Subtypen

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

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

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

TABELLEN ÜBER ALLGEMEINEN KLASSEN

- eine Tabelle für alle geographischen Objekte

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

Anfragen an Tabellen über allgemeinen Klassen

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

Anfragen an klassenspezifische Eigenschaften

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

Speziellste Klassenzugehörigkeit

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

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

type_name	typeid	supertype_name
mountain	08	geo_object

TYPTESTS

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

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

- wie bekommt man die Namen und die Höhe?

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

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

TYPUMWANDLUNGEN

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

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

9.5 Diverses zu Objekttypen

Änderungen an Objekttypen

Benutzerdefinierte Typen können mit `ALTER TYPE` verändert werden:

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

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

ÄNDERUNG VON TYPDEFINITIONEN: ABHÄNGIGKEITEN

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

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

- Funktions-basierte Indexe:

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

  arbeiten mit vorberechneten Werten.

ZUGRIFFSRECHTE AUF OBJEKTE

Recht an Objekttypen:

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

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

REFERENTIELLE INTEGRITÄT

- Vgl. `FOREIGN KEY ... REFERENCES ... ON DELETE/UPDATE CASCADE`
- Veränderungen an Objekten:
  OID bleibt unverändert
  $\rightarrow$ 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(50),
  Code       VARCHAR2(4),
  Capital    REF City_Type,
  Area       NUMBER,
  Population NUMBER);
/
```

Sinnvollerweise würde man hier gleich auch noch Methoden definieren.

```
CREATE OR REPLACE VIEW Country_ObjV OF Country_Type
WITH OBJECT OID (Code)
AS
SELECT Country.Name, Country.Code, REF(cty),  Area, Country.Population
FROM Country, City_ObjTab cty
WHERE cty.Name = Country.Capital AND cty.Province = Country.Province
  AND cty.Country = Country.Code;
```

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

OBJECT VIEWS: WAS NICHT GEHT

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

Object View auf Basis von *Organization_ObjTab*:

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

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

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

Beide angegebenen Attribute sind auch einzeln nicht erlaubt.

9.7 Objektorientierung und Data Dictionary

- TABS = USER_TABLES(table_name, ...): alle *relationalen* Tabellen die dem Benutzer gehören
- USER_OBJECT_TABLES(table_name, table_type, nested...): alle Objekttabellen die dem Benutzer gehören. `table_type` gibt den Objekttyp an, aus dessen Instanzen die Tabelle besteht. Hierzu gehören auch Nested Tables!
- USER_ALL_TABLES(table_name, ...): alle Tabellen die dem Benutzer gehören.
- USER_OBJECTS(object_name, object_type, status): alle Datenbankobjekte, die dem Benutzer gehören.
- Objekttabellen, deren Type mit DROP TYPE ... FORCE gelöscht wurde, stehen nicht mehr in USER_OBJECT_TABLES. Sie existieren aber noch (neues CREATE TABLE ergibt eine Fehlermeldung).
  (Bsp. nächste Folie)
- USER_TYPES(type_name): alle Objekttypen, die dem Benutzer gehören.

Löschen von Datenbankobjekten

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

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

- Verwendung von user_tables ignoriert invalide Tabellen.
- Nested Tables können nicht mit DROP TABLE gelöscht werden, sondern nur mit der Tabelle zu der sie gehören.
- Verwendung von "user_objects where object_type = 'TABLE'" bricht ab, wenn es eine (möglicherweise invalide, also in user_object_tables auch nicht mehr als nested gelistete) Nested Table löschen soll.

9.8 Fazit

- *Objektrelationale Tabellen* (Folie 256):
  Kompatibilität mit den grundlegenden Konzepten von SQL.
  U.a. Fremdschlüsselbedingungen von objektrelationalen Tabellen zu relationalen Tabellen.
- *Objektorientiertes Modell* (Folie 279):
  ... etwas kompliziert zu handhaben.
- *Object/Objekt-Relationale Views* (Folie 314):
  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 409), Multimedia etc.).

Kapitel 10
Embedded SQL

KOPPLUNGSARTEN ZWISCHEN DATENBANK- UND PROGRAMMIERSPRACHEN

- Erweiterung der Datenbanksprache um Programmierkonstrukte (z.B. PL/SQL)
- Erweiterung von Programmiersprachen um Datenbankkonstrukte: Persistente Programmiersprachen (Persistent Java – dann kein SQL als Anfragesprache)
- Datenbankzugriff aus einer Programmiersprache (JDBC)
- Einbettung der Datenbanksprache in eine Programmiersprache: “Embedded SQL” (C, Pascal, Java/SQLJ)

10.1 Embedded SQL: Grundprinzipien

... realisiert für C, Pascal, C++, Java (als SQLJ, siehe Folie 397) und weitere.

Impedance Mismatch bei der SQL-Einbettung

- Typsysteme Programmiersprache $\leftrightarrow$ Datenbanksystem passen nicht zusammen
- Unterschiedliche Paradigmen: Mengenorientiert vs. einzelne Variablen

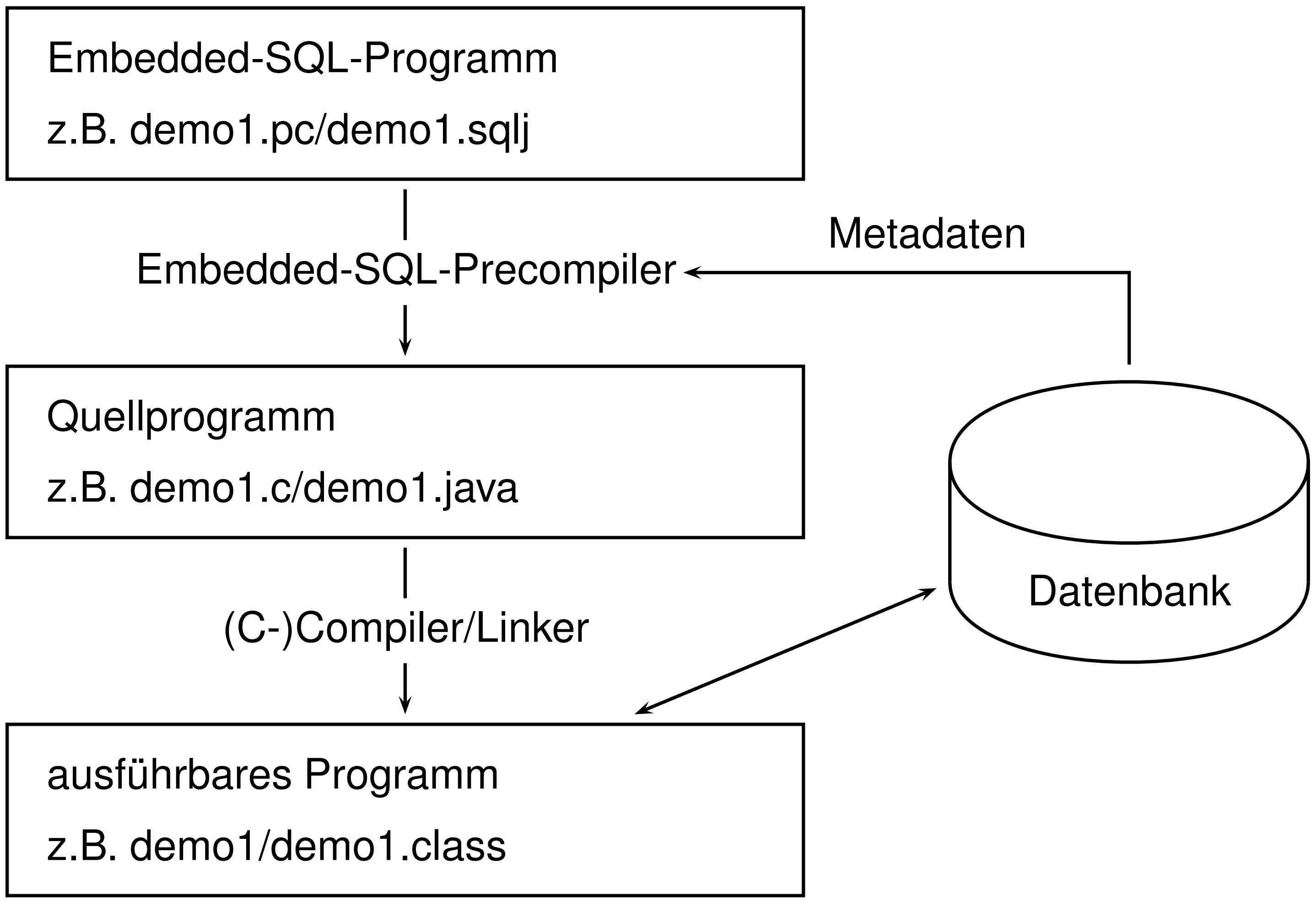
Realisierte Lösung

- Abbildung von Tupeln bzw. Attributen auf Datentypen der Hostsprache,
- Iterative Verarbeitung der Ergebnismenge mittels Cursor.

Auswirkungen auf die Hostsprache

- Struktur der Hostsprache bleibt unverändert,
- Spezielle Anweisungen zum Verbindungsaufbau,
- Jede SQL-Anweisung kann eingebettet werden,
- Verwendung von “Hostvariablen” (der umgebenden Programmiersprache) in SQL-Statements,
- SQL-Anweisungen wird `EXEC SQL` (oder sonstwas) vorangestellt.

ENTWICKLUNG EINER EMBEDDED SQL-APPLIKATION



- SQLJ (siehe Folie 397): Zwischenschritt bei der Compilierung muss nicht separat ausgeführt werden.

10.2 Embedded SQL in C [Legacy]

Hinweis: dieser Abschnitt kann ausgelassen und durch SQLJ (Folie 397) ersetzt werden. Er ist nur noch für die Arbeit mit Legacy-Datenbanken relevant, die diese Technologie verwenden.

VERBINDUNGSAUFBAU

Embedded-Anwendung: Verbindung zu einer Datenbank muss explizit hergestellt werden.

```
EXEC SQL CONNECT :username IDENTIFIED BY :passwd;
```

- `username` und `passwd` Hostvariablen vom Typ `CHAR` bzw. `VARCHAR`..
- Strings sind hier nicht erlaubt!

Äquivalent:

```
EXEC SQL CONNECT :uid;
```

wobei `uid` ein String der Form `"name/passwd"` ist.

HOSTVARIABLEN

- Kommunikation zwischen Datenbank und Anwendungsprogramm
- Output-Variablen übertragen Werte von der Datenbank zum Anwendungsprogramm
- Input-Variablen übertragen Werte vom Anwendungsprogramm zur Datenbank.
- jeder Hostvariablen zugeordnet: Indikatorvariable zur Verarbeitung von NULL-Werten.
- werden in der *Declare Section* deklariert:

```
EXEC SQL BEGIN DECLARE SECTION;
 int population;        /* host variable */
 short population\_ind;   /* indicator variable */
EXEC SQL END DECLARE SECTION;
```

- in SQL-Statements wird Hostvariablen und Indikatorvariablen ein Doppelpunkt (“:”) vorangestellt
- Datentypen der Datenbank- und Programmiersprache müssen kompatibel sein

Indikatorvariablen

Verarbeitung von Nullwerten und Ausnahmefällen

Indikatorvariablen für Output-Variablen:

- -1 : der Attributwert ist `NULL`, der Wert der Hostvariablen ist somit undefiniert.
- 0 : die Hostvariable enthält einen gültigen Attributwert.
- $>$0 : die Hostvariable enthält nur einen Teil des Spaltenwertes. Die Indikatorvariable gibt die ursprüngliche Länge des Spaltenwertes an.
- -2 : die Hostvariable enthält einen Teil des Spaltenwertes wobei dessen ursprüngliche Länge nicht bekannt ist.

Indikatorvariablen für Input-Variablen:

- -1 : unabhängig vom Wert der Hostvariable wird `NULL` in die betreffende Spalte eingefügt.
- $>=$0 : der Wert der Hostvariable wird in die Spalte eingefügt.

Cursore

- Analog zu PL/SQL
- notwendig zur Verarbeitung einer Ergebnismenge, die mehr als ein Tupel enthält

Cursor-Operationen

- `DECLARE <cursor-name> CURSOR FOR <sql statement>`
- `OPEN <cursor-name>`
- `FETCH <cursor-name> INTO <varlist>`
- `CLOSE <cursor-name>`

Fehlersituationen

- der Cursor wurde nicht geöffnet bzw. nicht deklariert
- es wurden keine (weiteren) Daten gefunden
- der Cursor wurde geschlossen, aber noch nicht wieder geöffnet

`Current of`**-Klausel** analog zu PL/SQL

Beispiel

```
int main() {
   EXEC SQL BEGIN DECLARE SECTION;
      char cityName[50];     /* output host var */
      int cityEinw;          /* output host var */
      char* landID = "D";    /* input host var */
      short ind1, ind2;      /* indicator vars */
      char* uid = "/";
   EXEC SQL END DECLARE SECTION;
   EXEC SQL CONNECT :uid;   /* Verbindung zur Datenbank herstellen */
   EXEC SQL DECLARE StadtCursor CURSOR FOR   /* Cursor deklarieren */
      SELECT Name, Einwohner  FROM Stadt  WHERE Code = :landID;
   EXEC SQL OPEN StadtCursor;  /* Cursor oeffnen */
   printf("Stadt                  Einwohner\n");
   while (1)
   { EXEC SQL FETCH StadtCursor INTO :cityName:ind1 ,
                         :cityEinw INDICATOR :ind2;
     if(ind1 != -1 && ind2 != -1)
     {  printf("%s     %d \n", cityName, cityEinw); }};  /* keine NULLwerte ausgeben *
   EXEC SQL CLOSE StadtCursor; }
```

HOSTARRAYS

- sinnvoll, wenn die Größe der Antwortmenge bekannt ist oder nur ein bestimmter Teil interessiert.
- vereinfacht Programmierung, da damit häufig auf einen Cursor verzichtet werden kann.
- verringert zudem Kommunikationsaufwand zwischen Client und Server.

```
EXEC SQL BEGIN DECLARE SECTION;
   char cityName[50][20];     /* host array */
   int cityPop[20];          /* host array */
EXEC SQL END DECLARE SECTION;
...
EXEC SQL SELECT Name, Population
         INTO :cityName, :cityPop
         FROM City
         WHERE Code = 'D';
```

holt 20 Tupel in die beiden Hostarrays.

PL/SQL IN EMBEDDED-ANWEISUNGEN

- Oracle Pro*C/C++ Precompiler unterstützt PL/SQL-Blöcke.
- PL/SQL-Block kann anstelle einer SQL-Anweisung verwendet werden.
- PL/SQL-Block verringt Kommunikationsaufwand zwischen Client und Server
- Übergabe in einem Rahmen:

```
EXEC SQL EXECUTE
DECLARE
  ...
BEGIN
  ...
END;
END-EXEC;
```

DYNAMISCHES SQL

SQL-Anweisungen können durch Stringoperationen zusammengestellt werden. Zur Übergabe an die Datenbank dienen unterschiedliche Befehle, abhängig von den in der Anweisung auftretenden Variablen.

TRANSAKTIONEN

- Anwendungsprogramm wird als geschlossene Transaktion behandelt, falls es nicht durch `COMMIT`- oder `ROLLBACK`-Anweisungen unterteilt ist
- In Oracle wird nach Beendigung des Programms automatisch ein `COMMIT` ausgeführt
- DDL-Anweisungen generieren vor und nach ihrer Ausführung implizit ein `COMMIT`
- Verbindung zur Datenbank durch
  `EXEC SQL COMMIT RELEASE;` oder
  `EXEC SQL ROLLBACK RELEASE;`
  beenden.
- Savepoints: `EXEC SQL SAVEPOINT <name>`

AUSNAHMEBEHANDLUNG: SQLCA (SQL COMMUNICATIONS AREA)

Enthält Statusinformationen bzgl. der zuletzt ausgeführten SQL-Anweisung

```
struct sqlca {
   char   sqlcaid[8];
   long   sqlcabc;
   long   sqlcode;
   struct { unsigned short sqlerrml;
            char sqlerrmc[70];
   } sqlerrm;
   char   sqlerrp[8];
   long   sqlerrd[6];
   char   sqlwarn[8];
   char   sqlext[8];
};
```

Interpretation der Komponente `sqlcode`:

- 0: die Verarbeitung einer Anweisung erfolgte ohne Probleme.
- $>$0: die Verarbeitung ist zwar erfolgt, dabei ist jedoch eine Warnung aufgetreten.
- $<$0: es trat ein ernsthafter Fehler auf und die Anweisung konnte nicht ausgeführt werden.

WHENEVER-Statement

spezifiziert Aktionen die im Fehlerfall automatisch vom DBS ausgeführt werden sollen.

```
EXEC SQL WHENEVER <condition> <action>;
```

`<condition>`

- `SQLWARNING` : die letzte Anweisung verursachte eine “no data found” verschiedene Warnung (siehe auch `sqlwarn`). Dies entspricht `sqlcode` $>$ 0 aber ungleich 1403.
- `SQLERROR` : die letzte Anweisung verursachte einen (ernsthaften) Fehler. Dies entspricht `sqlcode` $<$ 0.
- `NOT FOUND` : `SELECT INTO` bzw. `FETCH` liefern keine Antworttupel zurück. Dies entspricht `sqlcode` 1403.

`<action>`

- `CONTINUE` : das Programm fährt mit der nächsten Anweisung fort.
- `DO flq proc_name>` : Aufruf einer Prozedur (Fehlerroutine); `DO break` zum Abbruch einer Schleife.
- `GOTO <label>` : Sprung zu dem angegebenen Label.
- `STOP`: das Programm wird ohne `commit` beendet (exit()), stattdessen wird ein `rollback` ausgeführt.

Kapitel 11
Java und Datenbanken

- Java: plattformunabhängig
- überall, wo eine *Java Virtual Machine (JVM)* läuft, können Java-Programme ablaufen.
- APIs: Application Programming Interfaces; Sammlungen von Klassen und Schnittstellen, die eine bestimmte Funktionalität bereitstellen.

Mehrere der bisher behandelten Aspekte können mit Java gekoppelt werden:

- Prozeduren und Funktionen, Member Methods: Java Stored Procedures (Folie 337),
- Objekttypen: Java Object Types (Folie 343)
  (so kann man beliebige Datenstrukturen implementieren und anbieten → XML),
- Low-Level-Infrastruktur für Datenbankzugriff aus Java: JDBC (Folie 350),
- Embedded SQL (intern basierend auf JDBC): SQLJ (Folie 397).

STANDARDISIERUNG

- JDBC ist ein Java-API, das (DB-produktunabhängigen) low-level-Datenbankzugriff aus Java erlaubt.
- SQLJ: ANSI-Standard als Teil des SQL-Standards, bestehend aus drei Teilen:
  - Part 0: Embedded SQL in Java (ANSI X3.135.10-1998, bzw ISO-Standard “Database Language SQL:1999 – Part 10: Object Language Bindings (SQL/OLB)”; siehe Abschnitt “SQLJ”)
  - Part 1: SQL routines using Java (ANSI NCITS 331.1-1999, siehe Abschnitt “Java in Stored Procedures”).
  - Part 2: SQL types using Java (ANSI NCITS 331.2-2000, siehe Abschnitt “Java in SQL-Objekttypen”, u.a. $\rightarrow$ XMLType).
  - Part 1 und 2 bilden zusammen Part 13 des SQL:1999-Standards (ISO/IEC 9075-13:2002) “SQL Routines and Types Using the Java Programming Language (SQL/JRT)”

Technische Hinweise/Settings im IFI [Juli 2018]

- benötigte jars (in CLASSPATH reinnehmen):
  `/afs/informatik.uni-goettingen.de/group/dbis/public/oracle/instantclient` :
  ojdbc8.jar, [orai18n.jar], [orai18n-mapping.jar]
  `/afs/informatik.uni-goettingen.de/group/dbis/public/oracle/lib` :
  xdb.jar, [xmlparserv2.jar - Konflikt mit XML-P-jars]
  `/afs/informatik.uni-goettingen.de/group/dbis/public/oracle/sqlj/lib` :
  runtime12.jar, translator.jar
- Unix settings für Oracle (incl. CLASSPATH):
  `/afs/informatik.uni-goettingen.de/group/dbis/public/oracle/.oracle_env`
- Wenn JDBC-Verbindungsaufbau zu lange dauert:
  `java -Djava.security.egd=file:///dev/urandom` *`classfilename`*
  (JDBC-Verbindungen sind verschlüsselt. Normales /random basiert auf externen Ereignissen (Maus, Portanfragen), die an Poolrechnern nicht genügend vorliegen).

11.1 Java in Stored Procedures und Member Methods

- Oracle hat (seit 8i/8.1.5; Feb. 1999) eine eigene, integrierte JVM
  - keine GUI-Funktionalität
  - Java-Entwicklung außerhalb des DB-Servers
  - keine `main()`-Methode in Klassen, nur statische Methoden (= Klassenmethoden)
  - ab 9i Nutzung von Klassen als Objekttypen
  - kein Multithreading
  - DB-Zugriff über JDBC/SQLJ, dabei wird der serverseitige JDBC-Treiber verwendet (siehe Folien 350 und 397).
- Quelldateien (`.java`), Klassendateien (`.class`) oder Archive (`.jar`) können eingelesen werden.
- Shell: `loadjava`, `dropjava`
- DB: CREATE JAVA SOURCE, DROP JAVA SOURCE, DROP JAVA CLASS
- ```
  select object_type, object_name from user_objects
  where object_type like '%JAVA%' order by 1;
  ```
- Einbettung in Prozedur/Funktion (*Wrapper*, *call spec*) (void-Methoden als Prozeduren, non-void als Funktionen)
  ```

Laden von Java-Code per Shell

Außerhalb der DB wird eine Klasse geschrieben:

```
public class Greet
{ public static String sayHello (String name)
  { System.out.println("This is Java"); // Java output
    return "Hello " + name + "!";       // return value
 } }
```

[Filename: Java/Greet.java]

- Einlesen in die Datenbank mit `loadjava`.
  Empfehlenswert ist hier ein Alias um user/passwd nicht jedesmal angeben zu müssen:

  ```
  alias loadjava='loadjava -thin -u uname/passwd@dbis'
  dbis@s042> loadjava -r Greet.java
  ```

- `-r`: wird sofort compiliert und Referenzen aufgelöst (sonst: on-demand zur Laufzeit)
- Einlesen von .class-Dateien (ohne `-r`) – die Datenbank muss dieselbe Java-Version, wie auf dem Rechner wo man es compiliert hat, verwenden:

  ```
  dbis@s042> loadjava Greet.class
  ```

- analog mit .jar (das Sourcen und/oder class-Files enthält)
- Löschen einer Java-Klasse: analog mit `dropjava Greet` (auch hier alias definieren)

Erzeugen von Java-Klassen in SQL

Klasse ausserhalb der DB entwickeln und dann in der DB generieren:

```
CREATE OR REPLACE JAVA SOURCE NAMED "Hello"
AS
// here also imports are allowed
public class Greet
{ public static String sayHello (String name)
  { System.out.println("This is Java"); // Java output
    return "Hello " + name + "!";  // return value     }};
/
```
[Filename: Java/Greet-Create.sql]

- Wichtig: Name der Klasse in doppelte Anführungszeichen (case sensitive)
- mit dem als nächstes beschriebenen Wrapper eine PL/SQL-Prozedur daraus erzeugen,
- Löschen mit

  ```
  DROP JAVA SOURCE "Hello";
  ```

- Analog: Klassen als Binaries laden:

  ```
  CREATE OR REPLACE JAVA CLASS USING       BFILE(directory_object, filename);
  CREATE OR REPLACE JAVA CLASS USING       {BLOB|CLOB|BFILE} subquery;
  ```

Einbinden des Java-Codes in PL/SQL-Funktion/Prozedur

Innerhalb der Datenbank:

- Funktion als Wrapper (*call spec*):

```
CREATE OR REPLACE FUNCTION greet (person IN VARCHAR2)
  RETURN VARCHAR2 AS
LANGUAGE JAVA
NAME 'Greet.sayHello (java.lang.String)
      return java.lang.String';
/
```

[Filename: Java/Greet.sql]

- Bei void-Methoden: Prozedur als Wrapper
- Aufruf: `SELECT greet('Jim') FROM DUAL;`

GREET('JIM')
Hello Jim!

- Für die Java-Ausgabe muss man sowohl das Java-Output-Buffering als auch den SQL-Output aktivieren:

```
CALL dbms_java.set_output(2000);
SET SERVEROUTPUT ON;
```

Beispiel: `SELECT greet(name) FROM COUNTRY;`

Syntax des Prozedur/Funktions-Wrappers

```
CREATE [OR REPLACE]
  { PROCEDURE <proc_name>[(<parameter-list>)]
    | FUNCTION <func_name>[(<parameter-list>)] RETURN sql_type}
  {IS | AS} LANGUAGE JAVA
  NAME '<java_method_name>[(<java-parameter-list>)]
    [return <java_type_fullname>]';
  /
```

- Bei void-Methoden: Prozeduren,
- Bei non-void-Methoden: Funktionen,
- Die <parameter-list> muss der <java-parameter-list> entsprechen:
  - gleiche Länge,
  - sich entsprechende Parameter-Typen; Parameter-Typ-Mapping: siehe JDBC
- Achtung: In der NAME-Spezifikation muss `return` klein geschrieben werden,
- Aufruf des Wrappers eingebettet aus SQL und PL/SQL in Anfragen, DML-Operationen, Prozeduren, Triggern, ...

Soweit ist noch kein Datenbank-Zugriff aus den Methoden möglich. Dies wird durch JDBC ermöglicht (siehe Folie 350).

Nullwerte an Java übergeben

- wenn das Argument NULL ist, ist es in Java `null`,
- `return null` wird als SQL NULL-Wert interpretiert.

```
CREATE OR REPLACE JAVA SOURCE NAMED "Hello"
AS
public class Greet
{ public static String sayHello (String name)
  { System.out.println("This is Java");
    if (name != null)
       return "Hello " + name + "!";
    else return null;
} };
/
```

[Filename: Java/Greet-Null-Create.sql]

- wie vorher per Wrapper eine PL/SQL-Prozedur daraus erzeugen,
- `SELECT greet(NULL) FROM DUAL;`
- Anmerkung: in Oracle problemlos, in DB2 muss CREATE PROCEDURE mit GENERAL WITH NULLS bzw. SIMPLE WITH NULLS spezifiziert werden (→ Doku)

11.2 Java in SQL-Objekttypen

Man SQL-Typen auf Basis von Java-Klassen definieren. Die Java-Klasse muss das Interface `java.sql.SQLData` sowie ggf. weitere anwendungsspezifische Methoden implementieren:

- `public String getSQLTypeName()`
  liefert den entsprechenden SQL-Datentyp zurück
  "This method is called by the JDBC driver to get the name of the UDT [user-defined datatype] instance that is being mapped to this instance of SQLData."

- ```
  public void readSQL(SQLInput stream,
        String typeName) throws SQLException
  ```
  liest Daten aus der Datenbank und initialisiert das Java-Objekt

- `public void writeSQL(SQLOutput stream)`
  bildet das Java-Objekt auf die Datenbank ab.
  (vgl. Marshalling/Unmarshalling zwischen XML und Java in JAXB)

Diese drei Methoden werden nachher nicht vom Benutzer, sondern intern bei der Umsetzung aufgerufen.
  ```

BEISPIEL: JAVA-KLASSE GeoCoordJ

- Die Java-Klasse kann einen beliebigen Namen haben
- sie muss/soll die folgenden Methoden besitzen (siehe nächste Folie):
  - getSQLTypeName `MYGEOCOORD`: Name des SQL-Typs in Oracle
  - die Felddeklarationen (Namen können anders als in SQL sein)
  - Felder lesen/setzen in der Reihenfolge der SQL-Definition
  - Lese-/Schreibmethoden: `stream.read/write<type>`
  - Wenn die Klasse ausserdem in Java (mit JDBC) verwendet werden soll, ist ein Konstruktor und ein toString() sinnvoll.
  - die anwendungsspezifischen Methoden, die in Anfragen benutzt werden sollen (Namen können anders als in SQL sein).
- Klasse GeoCoordJ in Oracle laden :

```
dbis@s042> loadjava -r GeoCoordJ.java
```

- jetzt hat man eine Klasse, die man zur Definition eines SQL-Typs verwenden kann.

Beispiel: Java-Klasse `GeoCoordJ`

```
import java.sql.*;

public class GeoCoordJ implements java.sql.SQLData {
  private double lat, lon;

  public String getSQLTypeName() {
    return "MYGEOCOORD";   // just to illustrate something
 }
 public void readSQL(SQLInput stream, String typeName)
   throws SQLException {
     lat = stream.readDouble();
     lon = stream.readDouble();
 }
 public void writeSQL(SQLOutput stream)
   throws SQLException {
     stream.writeDouble(lat);
     stream.writeDouble(lon);
 }
 // ... continue next slide
```

```
// the constructor is not needed in the database
// but we need it for "outside" with JDBC
public GeoCoordJ(double la, double lo) {
   lat = la;  lon = lo;
}
// then, the standard constructor must be explicitly
// defined because it is needed in the database
public GeoCoordJ() {
   lat = 0;  lon = 0;
}
// used when printing e.g. in JDBC
public String toString() {
  return "GeoCoord(" + lat + "/" + lon + ")";
}
public double distance(GeoCoordJ other) {
  return
    6370 *  Math.acos(
       Math.cos(this.lat/180*3.14) *
       Math.cos(other.lat/180*3.14) *
       Math.cos( (this.lon - other.lon)
                 /180*3.14 ) +
       Math.sin(this.lat/180*3.14) *
       Math.sin(other.lat/180*3.14)  );
} }
```

[Filename: Java/GeoCoordJ.java]

SQL-WRAPPER-TYPE

- Namen des Typs, der Attribute/Spalten und der Methoden müssen nicht mit der verwendeten Java-Klasse übereinstimmen (EXTERNAL NAME ...):

```
CREATE OR REPLACE TYPE jgeocoord          -- Typname, der in SQL verwendet wird
AS OBJECT
EXTERNAL NAME 'GeoCoordJ  '          -- Name der Java-Klasse, die verwendet wird
LANGUAGE JAVA
USING SQLData
( latitude NUMBER EXTERNAL NAME 'lat',  -- Mapping der Attributnamen SQL<->Java
  longitude NUMBER EXTERNAL NAME 'lon',
  MEMBER FUNCTION distance (other IN jgeocoord) RETURN NUMBER   -- Signatur SQL
    EXTERNAL NAME 'distance (GeoCoordJ) return double');    -- Signatur in Java
/
CREATE TABLE jcoordtable OF jgeocoord;
INSERT INTO jcoordtable VALUES (jgeocoord(50,0));
INSERT INTO jcoordtable VALUES (jgeocoord(-34,151));

SELECT x.longitude, x.distance(jgeocoord(51.5,0))
FROM jcoordtable x;
```

LONGITUDE	X.DISTANCE(...)
0	166.681667
151	16977.5482

[Filename: Java/GeoCoordJ.sql]

Java-Objekte als Zeilenobjekte

- Signatur wie im CREATE TYPE vorgegeben.
  Diese wird wie optional mit den EXTERNAL NAME-Spezifikationen angegeben auf die Java-Klasse abgebildet.
- Tabelle sieht aus wie eine normale Tupel-/Objekttabelle:

```
DESC jcoordtable;
```

Name	Null?	Type
LATITUDE		NUMBER
LONGITUDE		NUMBER

- Einträge sehen aus wie ganz normale Zeilenobjekte:

```
SELECT * FROM jcoordtable;
```

Longitude	Latitude
50	0
-34	151

Java-Objekte als Spaltenobjekte

- duplicate the “Mountain” table with Java coordinates:
  recall: the name of the SQL type is ’jgeocoord’

```
CREATE TABLE mountainJcoord
AS (SELECT name, mountains, elevation, type,
       jgeocoord(m.coordinates.latitude,
                 m.coordinates.longitude) as coords
    FROM mountain m);

SELECT name, coords,
       m.coords.distance(jgeocoord(51.5,0))
FROM mountainJcoord m
ORDER BY 3 DESC;
```

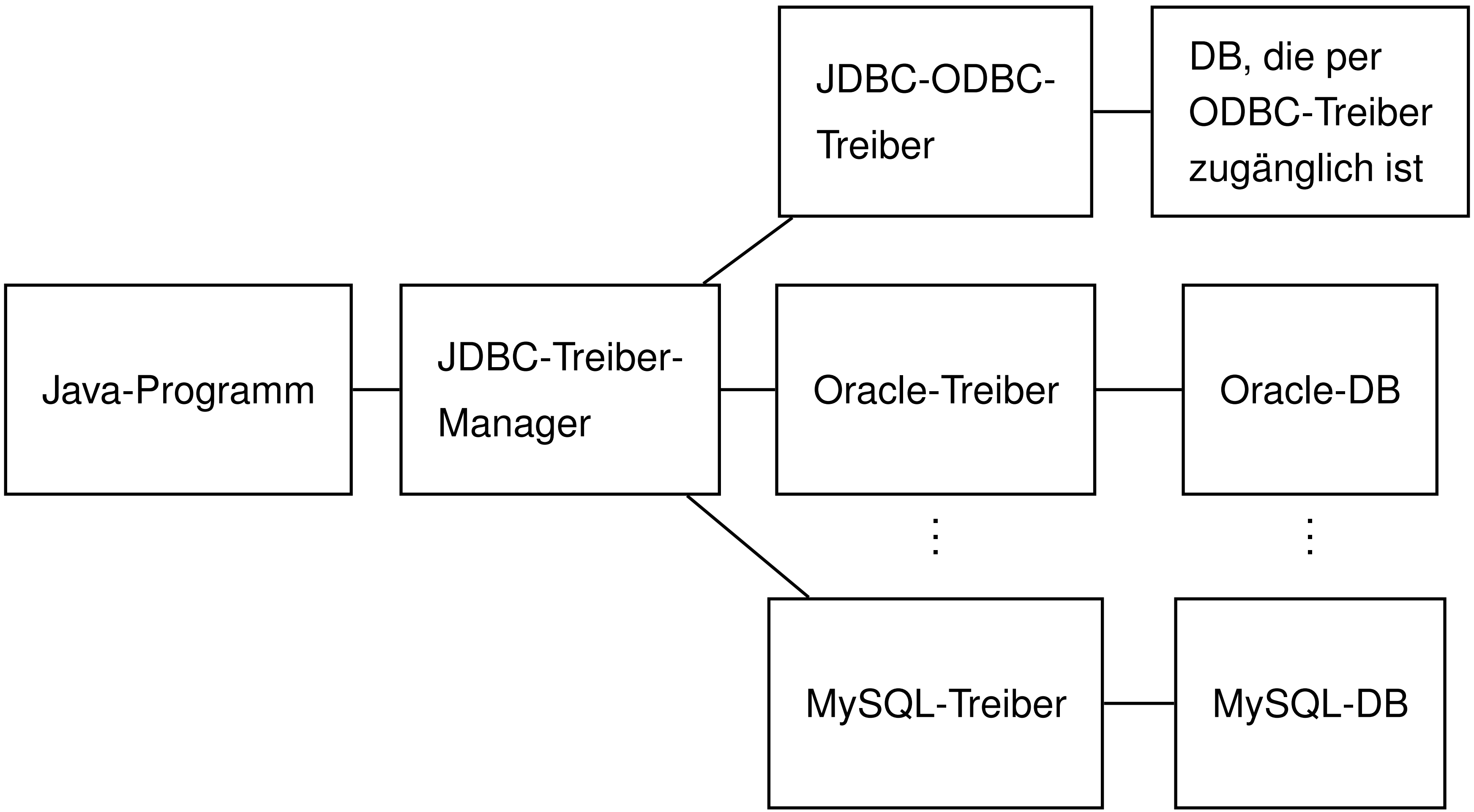
[Filename: Java/mountainjcoord.sql]

- `select * from mountainjcoord` :
  Einträge sind Spaltenobjekte, die z.B. als `JGEOCOORD(15.14, 120.3)`
  (d.h., der Typname der im CREATE TYPE genannt ist)
  ausgegeben werden.

11.3 JDBC (*Java Database Connectivity*): API für Low-Level-Datenbankzugriff

- Interface für den (entfernten) Datenbankzugriff von Java-Programmen aus,
- Teil des SDK (`java.sql.*`),
- Applikation kann unabhängig vom darunterliegenden DBMS programmiert werden,
- setzt die Idee von ODBC (Open DataBase Connectivity; ein 1992 entwickelter Standard zum Zugriff auf Datenbanken aus Programmiersprachen) auf Java um,
- gemeinsame Grundlage ist der X/Open SQL CLI (Call Level Interface) Standard.

JDBC-ARCHITEKTUR



JDBC-ARCHITEKTUR

- Kern: Treiber-Manager (`java.sql.DriverManager`)
- darunter: Treiber für einzelne DBMS'e

JDBC-API

- flexibel:
  - Applikation kann unabhängig vom darunterliegenden DBMS programmiert werden
- "low-level":
  - Statements werden durch Strings übertragen
  - im Gegensatz zu SQLJ (später) keine Verwendung von Programmvariablen in den SQL-Befehlen
    (d.h. Werte müssen explizit eingesetzt werden)

Darauf aufbauend:

- Embedded SQL für Java (SQLJ)
- direkte Darstellung von Tabellen und Tupeln in Form von Java-Klassen

JDBC-Funktionalität

- Aufbau einer Verbindung zur Datenbank (`DriverManager, Connection`)
- Versenden von SQL-Anweisungen an die Datenbank (`Statement`, `PreparedStatement` und `CallableStatement`)
- Verarbeitung der Ergebnismenge (`ResultSet`)

JDBC-TREIBER-MANAGER

`java.sql.DriverManager`

- verwaltet (registriert) Treiber
- wählt bei Verbindungswunsch den passenden Treiber aus und stellt Verbindung zur Datenbank her.
- Es wird nur ein `DriverManager` benötigt.

⇒ Klasse `DriverManager`:
  - nur `static` Methoden (operieren auf Klasse)
  - Konstruktor ist `private` (keine Instanzen erzeugbar)

Benötigte Treiber müssen angemeldet werden:

`DriverManager.registerDriver(driver*)`

Im Praktikum für den Oracle-Treiber:

```
DriverManager.registerDriver
      (new oracle.jdbc.driver.OracleDriver());
```

erzeugt eine neue Oracle-Treiber-Instanz und “gibt” sie dem DriverManager.

VERBINDUNGSAUFBAU

- `DriverManager` erzeugt offene Verbindungs-Instanz:

  ```
  Connection conn = DriverManager.getConnection(<jdbc-url>, <user-id>, <passwd>);
  ```

  oder

  ```
  Connection conn = DriverManager.getConnection(<jdbc-url>, <props-filename>);
  ```

  (Login-Daten aus Datei, via `java.util.Properties`).

- Datenbank wird eindeutig durch die JDBC-URL bezeichnet:
  - `jdbc:<subprotocol>:<subname>`
  - `<subprotocol>`: Name des produktspezifischen Treiber-Protokolls
  - `<subname>` ist produktspezifisch zusammengesetzt

  Bei uns, mit einer Oracle-Datenbank:

  ```
  jdbc:oracle:<driver-name>:@//<IP-Address DB Server>:<Port>/<Service>
  ```

  ```
  String url =
    'jdbc:oracle:thin:@//xxx.xxx.xxx.xxx:1521/dbis.informatik.uni-goettingen.de';
  ```

  (die aktuelle URL steht hoffentlich auf dem Aufgabenblatt)

- Verbindung beenden: `conn.close();`

VERSENDEN VON SQL-ANWEISUNGEN

`Statement`-Objekte

- werden durch Aufruf von Methoden einer bestehenden Verbindung `<connection>` erzeugt.
- `Statement`: einfache SQL-Anweisungen ohne Parameter
- `PreparedStatement`: Vorcompilierte Anfragen, Anfragen mit Parametern
- `CallableStatement`: Aufruf von gespeicherten Prozeduren

KLASSE "STATEMENT"

```
Statement <name> = <connection>.createStatement();
```

Sei `<string>` ein SQL-Statement *ohne Semikolon*.

- `ResultSet <statement>.executeQuery(<string>)`:
  SQL-Anfragen an die Datenbank. Dabei wird eine Ergebnismenge zurückgegeben.
- `int <statement>.executeUpdate(<string>)`: SQL-Statements, die eine Veränderung an der Datenbasis vornehmen (einschliesslich CREATE PROCEDURE etc). Der Rückgabewert gibt an, wieviele Tupel von der SQL-Anweisung betroffen waren.
- `boolean <statement>.execute(<string>)` – Sonstiges:
  - Generierung und Aufrufe von Prozeduren/Funktionen (siehe CallableStatements),
  - Statement dynamisch als String erzeugt, und man weiß nicht, ob es eine Query oder ein Update ist,
  - "true" wenn das (erste) Ergebnis ein ResultSet ist; "false" sonst (siehe später).

Ein `Statement`-Objekt kann beliebig oft wiederverwendet werden, um SQL-Anweisungen zu übermitteln.

Mit der Methode `close()` kann ein Statement-Objekt geschlossen werden.

BEHANDLUNG VON ERGEBNISMENGEN

Klasse “`ResultSet`” (Iterator-Pattern):

```
ResultSet <name> = <statement>.executeQuery(<string>);
```

- virtuelle Tabelle, auf die von der “Hostsprache” – hier also Java – zugegriffen werden kann.
- `ResultSet`-Objekt unterhält einen Cursor, der mit

  ```
  <result-set>.next();
  ```

  auf das nächste (bzw. am Anfang auf das erste) Tupel gesetzt wird.
- `<result-set>.next()`
  liefert den Wert `false` wenn alle Tupel gelesen wurden.

```
ResultSet countries =
  stmt.executeQuery("SELECT Name, Code, Population
                     FROM Country");
```

Name	**code**	**Population**
Germany	D	83536115
Sweden	S	8900954
Canada	CDN	28820671
Poland	PL	38642565
Bolivia	BOL	7165257
..	..	..

Behandlung von Ergebnismengen

- Zugriff auf die einzelnen Spalten des Tupels unter dem Cursor mit

  `<result-set>.get<type>(<attribute>)`

- `<type>` ist dabei ein Java-Datentyp,

SQL-Typ	get-Methode
INTEGER	getInt
REAL, FLOAT	getFloat
BIT	getBoolean
CHAR, VARCHAR	getString

  - `getString` funktioniert immer (*type casting*).
  - `getObject` funktioniert immer, dann mit `instanceof` prüfen und casten.

- `<attribute>` kann entweder durch Attributnamen oder durch die Spaltennummer gegeben sein.

```
countries.getString("Code");
countries.getInt("Population");
countries.getInt(3);
```

- Bei `get<type>` werden die Daten des Ergebnistupels (SQL-Datentypen) in Java-Typen konvertiert.

Beispiel-Code

```
import java.sql.*;
class JdbcCities {
public static void main (String args []) throws SQLException
{ // ORACLE-Treiber laden
  DriverManager.registerDriver (new oracle.jdbc.driver.OracleDriver());
  // Verbindung zur Datenbank herstellen
  String url = "jdbc:oracle:thin://@xxx.xxx.xxx.xxx:1521/dbis.informatik.uni-goettingen.de";
  Connection conn = DriverManager.getConnection(url,"scott","tiger");
  // Anfrage an die Datenbank
  Statement stmt = conn.createStatement();
  ResultSet rset = stmt.executeQuery("SELECT Name, Population FROM City");
  while (rset.next()) { // Ergebnis verarbeiten
    String s = rset.getString(1);
    int i = rset.getInt("Population");
    System.out.println (s + " " + i + "\n");
    }
  rset.close(); stmt.close(); conn.close();
  }}
```

Verbindungsaufbau zu Oracle mit Properties-File

- `conn.props` (Dateiname ist frei wählbar) ist eine Datei (bzw. Datei im .jar-Archiv), die folgendermassen aussieht:

```
#conn.props:
## fuer normales JDBC
url=jdbc:oracle:thin:@//xxx.xxx.xxx.xxx:1521/dbis.informatik.uni-goettingen.de
user=scott
password=tiger
### fuer SQLJ dasselbe nochmal
sqlj.url=jdbc:oracle:thin:@//xxx.xxx.xxx.xxx:1521/dbis.informatik.uni-goettingen.de
sqlj.user=scott
sqlj.password=tiger
```

[Filename: Java/conn.props – muss jeder selber schreiben]

- Eine Datei dieser Form kann man in eine Instanz von java.util.Properties einlesen (ein key-Value-Hash) und dann mit getProperty() oder automatisch abfragen.
- `conn.props` ggf. mit ins .jar legen.

Verbindungsaufbau mit Properties-File

```
import java.sql.*;
import java.io.FileInputStream;
import java.util.Properties;
class JdbcCities {
public static void main (String args []) throws Exception   // File oder SQL Exc
{ // Oracle-Treiber laden
  DriverManager.registerDriver (new oracle.jdbc.driver.OracleDriver());
  // Verbindung zur Datenbank herstellen
  Properties props = new Properties();
  props.load(new FileInputStream("conn.props"));
  Connection conn = DriverManager.getConnection(props.getProperty("url"), props);
   // Anfrage an die Datenbank
  Statement stmt = conn.createStatement();
  ResultSet rset = stmt.executeQuery("SELECT Name, Population FROM City");
  while (rset.next()) {     // Verarbeitung der Ergebnismenge
    String s = rset.getString(1);
    int i = rset.getInt("Population");
    System.out.println (s + " " + i);   }
    rset.close(); stmt.close(); conn.close(); }}
```

[Filename: Java/JdbcCities.java]

Behandlung von Ergebnismengen: JDBC-Datentypen

- JDBC steht zwischen Java (Objekttypen) und SQL (Typen mit unterschiedlichen Namen).
- `java.sql.Types` definiert *generische* SQL-Typen, mit denen JDBC arbeitet:
  (oft Subklassen der Java-Typen)

Java-Typ	JDBC-SQL-Typ in `java.sql.Types`
`java.lang.String`	`CHAR, VARCHAR`
`java.math.BigDecimal`	`NUMBER, NUMERIC, DECIMAL`
`boolean`	`BIT`
`byte`	`TINYINT`
`short`	`SMALLINT`
`int`	`INTEGER`
`long`	`BIGINT`
`float`	`REAL`
`double`	`FLOAT, DOUBLE`
`convert to java.time.*`	`TIMESTAMP (DATE, TIME)`

Diese werden auch verwendet, um Meta-Daten zu verarbeiten.

BEHANDLUNG VON ERGEBNISMENGEN

Im Fall von allgemeinen Anfragen weiß man oft nicht, wieviele Spalten eine Ergebnismenge hat, wie sie heißen, und welche Typen sie haben.

Instanz der Klasse `ResultSetMetaData` enthält Metadaten über das vorliegende ResultSet:

```
ResultSetMetaData <name> = <result-set>.getMetaData();
```

erzeugt ein `ResultSetMetaData`-Objekt, das Informationen über die Ergebnismenge enthält:

- `int getColumnCount()`:
  Spaltenanzahl der Ergebnismenge
- `String getColumnLabel(int)`:
  Attributname der Spalte ‹int›
- `String getTableName(int)`:
  Tabellenname der Spalte ‹int›
- `int getColumnType(int)`:
  JDBC-Typ der Spalte ‹int›
  (ein SQL Type aus der Enumeration java.sql.Types; 2002 ist java.sql.Types.STRUCT)
- `String getColumnTypeName(int)`:
  Unterliegender DBMS-Typ der Spalte ‹int›

Behandlung von Ergebnismengen

- keine NULL-Werte in Java:

  `<resultSet>.wasNull()`

  testet, ob der zuletzt gelesene Spaltenwert NULL war.

Beispiel: Ausgabe aller Zeilen eines ResultSets

```
ResultSetMetaData rsetmetadata = rset.getMetaData();
int numCols = rsetmetadata.getColumnCount();
while (rset.next()) {
  for(int i=1; i<=numCols; i++) {
    String returnValue = rset.getString(i);
    if (rset.wasNull())
       System.out.println ("null");
    else
       System.out.println (returnValue);
  }
}
```

- Mit der Methode `close()` kann ein ResultSet-Objekt explizit geschlossen werden.

Beispiel: Auslesen einer beliebigen Tabelle

```
import java.sql.*;
import java.io.FileInputStream;
import java.util.Properties;
class JdbcSelect {
  public static void main (String args []) throws Exception {
  DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
  Properties props = new Properties();
  props.load(new FileInputStream("conn.props"));
  Connection conn = DriverManager.getConnection(props.getProperty("url"), props);
  Statement stmt = conn.createStatement();
  ResultSet rset = stmt.executeQuery("SELECT * FROM " + args[0]);
   ResultSetMetaData rsetmetadata = rset.getMetaData();
  int numCols = rsetmetadata.getColumnCount();
  while (rset.next()) {
    for(int i=1; i<=numCols; i++) {
      String value = rset.getString(i);
      if (rset.wasNull()) System.out.print("null");
         else System.out.print(value);
      System.out.print("  ");
    }
    System.out.println();
  }
  rset.close(); stmt.close(); conn.close();
}}
```

```
dbis@c42> java JdbcSelect City
```

[Filename: Java/JdbcSelect.java]

Auslesen von einfachen Zeilenobjekten

Zeilenobjekte entsprechen Tupeln über atomaren Datentypen (vgl. Folie 348):

```
dbis@c42> java JdbcSelect jcoordtable
```

Auslesen von Objekten: benutzerdefinierte SQL-Typen

- Für Instanzen von Spaltenobjekten wird "null" ausgegeben, oder man erhält sogar einen Fehler:

  ```
  dbis@c42> java jdbcSelect mountain
  ```

- Objekte mit getObject(n) auslesen
  (behandelt auch Strings, Zahlen etc. korrekt)
- Instanzen von mit PL/SQL benutzerdefinierten SQL-Typen haben als SQL-Typ (aus den Metadaten mit ResultSetMetaData.getColumnType(i)) java.sql.Types.STRUCT und implementieren das Interface java.sql.STRUCT.
- Objekttyp-Name (aus der Instanz): String name = x.getSQLTypeName()
- attribute: Object[ ] attrs = x.getAttributes()
  enthält dann Strings, Zahlwerte, oder wieder Objekte

Beispiel: Auslesen von SQL-Objekten – Metadaten

- erster Teil: Metadaten der Spalten aus dem ResultSetMetaData:
  - JDBC-Typen aus der Enum java.sql.Types
    z.B. 12=VARCHAR, 2=NUMERIC, 2002=STRUCT, (Oracle: 2008 = Java-definiert)
  - Typnamen im DBMS
    z.B. das PL/SQL-definierte MAY.GEOCOORD oder das Java-definierte MAY.JGEOCOORD

```
import java.sql.*;
import java.io.FileInputStream;
import java.util.Properties;
class JdbcSelectObj {
  public static void main (String args []) throws Exception {
   Connection conn = getConn();  // unten rausgescrollt
   Statement stmt = conn.createStatement();
   ResultSet rset = stmt.executeQuery("SELECT * FROM " + args[0]);
   ResultSetMetaData rsmd = rset.getMetaData();
   int numCols = rsmd.getColumnCount();
   for(int i=1; i<=numCols; i++)  System.out.print(rsmd.getColumnType(i) + " ");
   System.out.println();
   for(int i=1; i<=numCols; i++)  System.out.print(rsmd.getColumnTypeName(i) + " ");
   System.out.println();
   // continue next slide
```

```
  while (rset.next()) {
   for(int i=1; i<=numCols; i++) {
    Object value = rset.getObject(i);
    // System.out.println(rsmd.getColumnType(i));
    if (rset.wasNull()) System.out.print("null  ");
    else if (rsmd.getColumnType(i) == java.sql.Types.STRUCT)
    { java.sql.Struct s = (java.sql.Struct)value;
      System.out.print(s.getSQLTypeName() + "( ");
      Object[] attrs = s.getAttributes();
      // attributes
      for (int j = 0; j < attrs.length; j++) System.out.print(attrs[j] + " ");
      System.out.print(")");
    }
    // This also covers Java instances (uses their toString())
    else System.out.print(value + "  ");
  }
  System.out.println();  }
  rset.close(); stmt.close(); conn.close();  }
private static Connection getConn() throws Exception {
  DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
  Properties props = new Properties();
  props.load(new FileInputStream("conn.props"));
  Connection conn = DriverManager.getConnection(props.getProperty("url"), props);
  return conn;  }}
```

[Filename: Java/JdbcSelectObj.java]

Auslesen von Objekten von PL/SQL-Typen

- gibt die Instanzen von via PL/SQL definierten Objekttypen generisch als STRUCTS, Kontruktoraufruf und Parameter (ohne Komma zwischen den Parametern) aus:

  `dbis@c42> java jdbcSelectObj mountain`

  Ausgabe der Koordinaten z.B. als `MAY.GEOCOORD(19.2 -98.6)`.

Auslesen von Objekten: Instanzen von Java-Klassen

Für Instanzen von Java-basierten SQL-Typen (jgeocoord) in Spaltenobjekten (d.h. nicht für die Zeilenobjekte von Folie 348, die wie Tupel behandelt werden) wird eine Instanz der entsprechenden Java-Klasse erzeugt:

- (Folie 347): SQL-Typ "jgeocoord" benutzt die Java-Klasse "GeoCoordJ".
- GeoCoordJ.java/.class liegt zugreifbar im "Java/"-Subdirectory.
- sonst: java.sql.SQLException: ... ClassNotFoundException: GeoCoordJ

⇒ Bei der Konvertierung des SQL-Typs aus dem JDBC-ResultSet ist der (abweichende!) Name der Java-Klasse bekannt, der Java-Code aber nicht (würde evtl. auch auf der JVM des Clients garnicht laufen).

- Wenn man die "Original-Klasse", die in der DB verwendet wird, nicht hat, kann man dort irgendeine Klasse hinlegen, die den entsprechenden Namen hat und die Methoden des Interfaces java.sql.SQLData und diejenigen Methoden, die man tatsächlich benutzen möchte (auch das können mehr sein, als im Original), implementiert.
- `java JdbcSelectObj mountainJCoord`
  ruft generisch im normalen print-Zweig deren toString() auf.

⇒ kein cast notwendig, da für den Compiler toString() bei Object definiert ist, und der Code *zur Laufzeit* dynamisch korrekt gebunden wird.

Auslesen und Verwenden von Java-Objekten

- Wenn man Methoden/Felder einer Instanz einer Java-Klasse verwenden will, die bei Object nicht automatisch definiert sind, muss man entsprechend testen (Java-Typen haben als java.sql.Types bei Oracle die Konstante 2008) und explizit casten.

```
import java.sql.*;
import java.io.FileInputStream;
import java.util.Properties;
class JdbcSelectObjCast {
  public static void main (String args []) throws Exception {
    DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
    Properties props = new Properties();
    props.load(new FileInputStream("conn.props"));
    Connection conn = DriverManager.getConnection(props.getProperty("url"), props);
    Statement stmt = conn.createStatement();
    ResultSet rset = stmt.executeQuery("SELECT * FROM " + args[0]);
    ResultSetMetaData rsmd = rset.getMetaData();
    int numCols = rsmd.getColumnCount();

    // continue next slide
```

```
while (rset.next()) {
 for(int i=1; i<=numCols; i++) {
  Object value = rset.getObject(i);
  // System.out.println(rsmd.getColumnType(i));
  if (rset.wasNull()) System.out.print("null");
  else if (rsmd.getColumnType(i) == 2008 &&
     ((java.sql.SQLData)value).getSQLTypeName() == "MYGEOCOORD")  // <<<< test
  { GeoCoordJ gc = (GeoCoordJ)value;         // <<<< cast
    System.out.print(gc + " ");
    System.out.print(gc.distance(new GeoCoordJ(0,0)));  // <<<< call
  }
  else if (rsmd.getColumnType(i) == 2008 && value instanceof GeoCoordJ) // <<<< test
  { GeoCoordJ gc = (GeoCoordJ)value;         // <<<< cast
    System.out.print(gc + " ");
    System.out.print(gc.distance(new GeoCoordJ(0,0))); // <<<< call
  }
  else if (rsmd.getColumnType(i) == java.sql.Types.STRUCT)
  { System.out.print(((java.sql.Struct)value).getSQLTypeName() + "( ");
    Object[] attrs = ((java.sql.Struct)value).getAttributes();
    for (int j = 0; j < attrs.length; j++) System.out.print(attrs[j] + " ");
    System.out.print(")");
  }
  else System.out.print(value + "  "); // also covers Java instances (uses toString())
}
System.out.println();  }
rset.close(); stmt.close(); conn.close();  }}
```

DYNAMISCHES SQL: GENERISCHES EXECUTE()

(selten verwendet)

Häufig: `<string>` dynamisch generiert. Wenn man nicht weiß, ob es eine Query oder ein Update ist, kann man mit execute() generisch vorgehen.

- `boolean <statement>.execute(<string>)`,
  (nur für Statement, nicht für PreparedStatement und CallableStatement)
- "true" wenn das (erste) Ergebnis ein ResultSet ist; "false" sonst.
- `ResultSet getResultSet()`: Falls das (erste) Ergebnis eine Ergebnismenge ist, wird diese zurückgegeben; falls kein Ergebnis mehr vorhanden, oder das (erste) Ergebnis ein Update-Zähler ist: `null` zurückgeben.
- `int getUpdateCount()`: Falls das (erste) Ergebnis ein Update-Zähler ist, wird dieser ($n \geq 0$) zurückgegeben; falls kein Ergebnis mehr vorhanden, oder das (erste) Ergebnis eine Ergebnismenge ist, wird -1 zurückgegeben.

PREPARED STATEMENTS

```
PreparedStatement <name> =
  <connection>.prepareStatement(<string>);
```

- SQL-Anweisung `<string>` wird vorcompiliert.
- damit ist die Anweisung fest im Objektzustand enthalten
- effizienter als `Statement`, wenn ein SQL-Statement häufig (mit ggf. verschiedenen Parametern) ausgeführt werden soll.
- Abhängig von `<string>` ist nur eine der (parameterlosen!) Methoden
  - `<prepared-statement>.executeQuery()` oder
  - `<prepared-statement>.executeUpdate()`

  anwendbar.

Prepared Statements: Parameter

- Eingabeparameter werden durch “?” repräsentiert

```
PreparedStatement giveCountryPop =
  conn.prepareStatement("SELECT Population FROM Country WHERE Code = ?");
```

- “?”-Parameter werden mit

  `<prepared-statement>.set<type>(<pos>,<value>);`

  gesetzt, bevor ein `PreparedStatement` ausgeführt wird.

- `<type>`: Java-Datentyp,
- `<pos>`: Position des zu setzenden Parameters,
- `<value>`: Wert.

Beispielsequenz:

```
giveCountryPop.setString(1,"D");
ResultSet rset = giveCountryPop.executeQuery();
if (rset.next()) System.out.print(rset.getInt(1));
giveCountryPop.setString(1,"CH");
ResultSet rset = giveCountryPop.executeQuery();
if (rset.next()) System.out.print(rset.getInt(1));
```

PreparedStatement (Cont'd)

- Nullwerte werden gesetzt durch

  `setNULL(<pos>,<sqlType>);`

  `<sqlType>` bezeichnet den JDBC-Typ dieser Spalte.

- nicht sinnvoll in Anfragen (Abfrage nicht mit "= NULL" sondern mit "IS NULL"), sondern z.B. bei INSERT-Statements oder Prozeduraufrufen etc.

Beispiel: PreparedStatement

```
import java.sql.*;
import java.io.FileInputStream;
import java.util.Properties;
class JdbcCountryPop {
  public static void main (String args []) throws Exception {
 DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
 Properties props = new Properties();
 props.load(new FileInputStream("conn.props"));
 Connection conn = DriverManager.getConnection(props.getProperty("url"), props);
 PreparedStatement giveCountryPop =
   conn.prepareStatement("SELECT Population FROM Country WHERE Code = ?");
 giveCountryPop.setString(1,args[0]);
 ResultSet rset = giveCountryPop.executeQuery();
 if(rset.next()) {
  int pop = rset.getInt(1);
  if (rset.wasNull()) System.out.println("null");
  else System.out.println(pop);  }
 else System.out.println("No existing country code");
 conn.close();
}}
```

```
dbis@c42> java JdbcCountryPop D
dbis@c42> java JdbcCountryPop X
```

[Filename: Java/JdbcCountryPop.java]

ERZEUGEN VON FUNKTIONEN, PROZEDUREN ETC.

- Erzeugen von Prozeduren und Funktionen mit

  `<statement>.executeUpdate(<string>);`

  (`<string>` von der Form `CREATE PROCEDURE ...`)

```
s = 'CREATE PROCEDURE bla() IS BEGIN ... END';
stmt.executeUpdate(s);
```

CALLABLE STATEMENTS: GESPEICHERTE PROZEDUREN

Der *Aufruf der Prozedur* wird als `CallableStatement`-Objekt erzeugt:

- Aufrufsyntax von Prozeduren bei den verschiedenen Datenbanksystemen unterschiedlich

⇒ JDBC verwendet eine *generische* Syntax per Escape-Sequenz (Umsetzung dann durch Treiber)

```
CallableStatement <name> =
  <connection>.prepareCall("{call <procedure>}");
```

```
CallableStatement cstmt =
  conn.prepareCall("{call bla()}");
```

CALLABLE STATEMENTS MIT PARAMETERN

```
s = 'CREATE FUNCTION distance(city1 IN Name, city2 IN Name)
     RETURN NUMBER IS BEGIN ... END';
stmt.executeUpdate(s);
```

- Parameter:

```
CallableStatement <name> =
  <connection>.prepareCall("{call <procedure>(?,...,?)}");
```

- Rückgabewert bei Funktionen:

```
CallableStatement <name> =
  <connection>.prepareCall ("{? = call <procedure>(?,...,?)}");
cstmt = conn.prepareCall("{? = call distance(?,?)}");
```

- Für `OUT`-Parameter sowie den Rückgabewert muss zuerst der JDBC-Datentyp der Parameter mit

  ```
  <callable-statement>.registerOutParameter (<pos>,java.sql.Types.<type>);
  ```

  registriert werden.

  ```
  cstmt.registerOutParameter(1,java.sql.Types.NUMERIC);
  ```

CALLABLE STATEMENTS MIT PARAMETERN

- Vorbereitung (s.o.)

```
cstmt = conn.prepareCall("{? = call distance(?,?)}");
cstmt.registerOutParameter(1,java.sql.Types.NUMERIC);
```

- `IN`-Parameter werden über `set<type>` gesetzt:

```
cstmt.setString(2,"Gottingen");
cstmt.setString(3,"Berlin");
```

- Aufruf mit

```
cstmt.execute();
```

- Lesen des `OUT`-Parameters mit `get<type>`:

```
int distance = cstmt.getInt(1);
```

Beispiel: CallableStatement

```
import java.sql.*;
import java.io.FileInputStream;
import java.util.Properties;
class JdbcCallProc {
  public static void main (String args []) throws Exception {
 DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
 Properties props = new Properties();
 props.load(new FileInputStream("conn.props"));
 Connection conn = DriverManager.getConnection(props.getProperty("url"), props);

 CallableStatement call = conn.prepareCall("{? = call greet(?)}");
 call.registerOutParameter(1,java.sql.Types.VARCHAR);
 call.setString(2,args[0]);
 call.execute();
 String answer = call.getString(1);
 System.out.println(answer);
 conn.close();
}}
```

[Filename: Java/JdbcCallProc.java]

Wenn die Funktion “Greet” (vgl. Folie 340) für den User verfügbar ist:

```
dbis@c42> java JdbcCallProc Joe
```

FOLGE VON (UPDATE) STATEMENTS VERARBEITEN

- jedes Statement mit execute() einzeln abzuschicken kostet Zeit.

Statement.addBatch(string)/executeBatch()

- `<statement>.addBatch(<string>)`: Statement (keine Query, nur DML-Updates und DDL-Statements) zum Batch dazunehmen,
- ... beliebig oft ... und dann
- `int[] <statement>.executeBatch()`: alle Statements ausführen; ergibt ein Array mit updateCount-Werten,
- `clearBatch()`

⇒ Folge *verschiedener* Statements erzeugen und ausführen lassen.

PreparedStatement.executeBatch()

- mit `conn.prepareStatement(<string>)` mit Parameter-?-Liste erzeugen
- Parameter mit `<preparedStatement>.set<type>(<pos>,<value>)` setzen,
- `<preparedStatement>.addBatch()`: Gesetzte Werte zum Batch dazunehmen,
- ... beliebig oft ... und dann
- `int[] <preparedStatement>.executeBatch()`: Statement für alle Parametertupel ausführen; ergibt ein Array mit updateCount-Werten,
- `clearBatch()`

⇒ Folge desselben Statements mit verschiedenen Parametern ausführen lassen.

- `con.setAutoCommit(true)` (true ist Default) ist dann auch praktisch.

CallableStatement.executeBatch()

- analog.

ABFRAGEN VON ERGEBNISSEN BEI EXECUTE ()

Häufig: ‹string› dynamisch generiert.

- erste Annahme: liefert nur ein Ergebnis.
- `boolean <stmt>.execute(<string>)`,
  `boolean <prepstmt>.execute()`, `boolean <callstmt>.execute()`
- "true" wenn das (erste) Ergebnis ein ResultSet ist;
  "false" sonst.
- `ResultSet <stmt>.getResultSet()`: Falls das (erste) Ergebnis eine Ergebnismenge ist, wird diese zurückgegeben; falls kein Ergebnis mehr vorhanden, oder das (erste) Ergebnis ein Update-Zähler ist: null zurückgeben.
- `int <stmt>.getUpdateCount()`: Falls das (erste) Ergebnis ein Update-Zähler ist, wird dieser ($n \geq 0$) zurückgegeben; falls kein Ergebnis mehr vorhanden, oder das (erste) Ergebnis eine Ergebnismenge ist, wird -1 zurückgegeben.

FOLGE VON ERGEBNISSEN VERARBEITEN

SQL-Statements, die mehrere Ergebnisse nacheinander zurückliefern.

- `<stmt>.execute(<string>)`, wobei `<string>` mehrere durch ";" getrennte Statements enthält.
  - in Oracle 11 nicht erlaubt
  - in Postgres (und einigen anderen) erlaubt
- `boolean <stmt>.getMoreResults()`: schaltet zum nächsten Ergebnis. true, wenn das nächste Ergebnis eine Ergebnismenge ist, false, wenn es ein Update-Zähler ist, oder keine weiteren Ergebnisse.
- alle Ergebnisse verarbeitet:

  ```
  (( <stmt> .getResultSet() == null) &&
   ( <stmt> .getUpdateCount() == -1))
  ```

  bzw.

  ```
  (( <stmt> .getMoreResults() == false) &&
   ( <stmt> .getUpdateCount() == -1))
  ```

Folge von Ergebnissen verarbeiten

```
import java.sql.Connection;   import java.sql.DriverManager;
import java.sql.ResultSet;    import java.sql.SQLException;   import java.sql.Statement;

public class JdbcSequence {
 public static void main(String[] args){
  try { // sequence only allowed with postgres!
   Connection con = DriverManager.getConnection("jdbc:postgresql://localhost:5432/thedb", "user", "passwd
   String sql = "CREATE TABLE test AS SELECT * FROM continent; " +
                "DELETE FROM test; DROP TABLE test; " +
                "SELECT * FROM country;";
   Statement stmt = con.createStatement();
   boolean hasMoreResultSets = stmt.execute(sql);
   while ( hasMoreResultSets || stmt.getUpdateCount() != -1 ) {
     if ( hasMoreResultSets ) { // if has rs
        ResultSet rs = stmt.getResultSet();
         int size = 0; while(rs.next()){ size++; }
         System.out.println("Result set size: " + size); }
     else { // if ddl/dml/...
       int queryResult = stmt.getUpdateCount();
       System.out.println(queryResult + " tuples updated");
     } // check whether to continue in the loop:
     hasMoreResultSets = stmt.getMoreResults();
   } // while results
   stmt.close(); con.close();
  } catch (SQLException e) { e.printStackTrace(); } } }
```

[Filename: Java/JdbcSequence.java]

TRANSAKTIONSSTEUERUNG

Per Default ist für eine `Connection` der Auto-Commit-Modus gesetzt:

- implizites Commit nach jeder ausgeführten Anweisung (Transaktion besteht also nur aus einem Statement)
- `con.setAutoCommit(false)` schaltet den Auto-Commit-Modus aus und man muss explizite Commits ausführen.

Dann hat man die folgenden Methoden:

- `con.setSavepoint(String name)` (setzt Sicherungspunkt)
- `con.commit()` (macht Änderungen persistent)
- `con.rollback([<savepoint>)]` (nimmt alle Änderungen [bis zu `<savepoint>`] zurück.

Beispiel: Transaktionen in JDBC

```
import java.sql.*;    import java.io.FileInputStream;   import java.util.Properties;
class JdbcTransactions {
public static void main (String args []) throws Exception {
  DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
  Properties props = new Properties();
  props.load(new FileInputStream("conn.props"));
  Connection conn1 = DriverManager.getConnection(props.getProperty("url"), props);
  conn1.setAutoCommit(false);
  Statement stmt1 = conn1.createStatement();
  stmt1.execute("DROP TABLE TTEST");
  stmt1.execute("CREATE TABLE TTEST(A VARCHAR2(4))");
  stmt1.execute("COMMIT");
  stmt1.execute("INSERT INTO TTEST VALUES('1BLA')");
  System.out.println ("select from conn1:");
  ResultSet rset1 = stmt1.executeQuery("SELECT A FROM TTEST");
  while (rset1.next()) { String s = rset1.getString(1);  System.out.println (s); }
  Connection conn2 = DriverManager.getConnection(props.getProperty("url"), props);
  // Anfrage an die Datenbank
  Statement stmt2 = conn2.createStatement();
  stmt2.execute("INSERT INTO TTEST VALUES('2FOO')");
  System.out.println ("select from conn2:");
  ResultSet rset2 = stmt2.executeQuery("SELECT A FROM TTEST");
  while (rset2.next()) { String s = rset2.getString(1); System.out.println (s); }
  stmt1.execute("COMMIT"); conn1.close(); conn2.close();  } }
```

[Filename: Java/JdbcTransactions.java]

FEHLERBEHANDLUNG IN ANWENDUNGEN

- JDBC-Aufrufe werfen ggf. `SQLException`s.
- Nicht geschlossene Verbindungen bleiben offen.
- SQL-Ausführung in try-catch-Block mit finally einbetten:

```
Connection con = null;
Statement stmt = null;
ResultSet rset = null;
try {
      ... con, stmt, rset aufbauen und verarbeiten ...
    } catch (SQLException e) { e.printStackTrace(); }
      finally { rset.close(); stmt.close(); con.close(); }
```

FEHLERBEHANDLUNG, EFFIZIENZ

- Wenn die Java-Objekte conn, stmt, rset nur lokal sind, baut der Garbage-Collector sie auch schnell genug ab und schließt die Verbindungen.
- Bei häufigem Kontakt zu derselben Datenbank ist es effizienter, Connection-Objekte nur einmal zu erzeugen/aufzubauen und dann in einem Pool zu verwalten: org.apache.commons.pool.impl.GenericObjectPool‹T›
- `con.close()` gibt die Connection dann an den Pool zurück.
- In diesem Fall werden bei Fehlern liegengebliebene Connections ohne try-catch-finally nicht zurückgegeben, und bleiben offen.

⇒ bei 150 (default) Connections blockiert der Server:
java.sql.SQLRecoverableException: I/O-Fehler: Got minus one from a read call

Fehlerbehandlung - Demo

```
import java.sql.*;
import java.util.HashSet;  import java.io.FileInputStream;  import java.util.Properties;
class JdbcConnectionOverflow {
public static void main (String args [])
throws Exception {
  DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
  Properties props = new Properties();
  props.load(new FileInputStream("conn.props"));
  String url = props.getProperty("url");
  Connection conn;  Statement stmt;  ResultSet rset;
  int i = 0;
  HashSet<Connection> s = new HashSet<Connection>();
  while (true) {
    try { Thread.sleep(200);  i++;  System.out.println(i);
      conn = DriverManager.getConnection(url, props);
      s.add(conn);
      stmt = conn.createStatement();
      rset = stmt.executeQuery("select * from qwertz");
    } catch (SQLException e) { e.printStackTrace(); }
    catch (InterruptedException ex) { Thread.currentThread().interrupt(); }
}}}
/* ADMIN only:
select username, count(*) from V$SESSION group by username; */
```

[Filename: Java/JdbcConnectionOverflow.java]

TECHNISCHER HINWEIS FALLS JDBC-VERBINDUNGSAUFBAU STOCKT

Es kann vorkommen, dass JDBC-Aufrufe plötzlich lange dauern (und beim nächsten Versuch wieder schnell gehen usw.).

Siehe `https://blog.dbi-services.com/connect-times-to-the-db-suddenly-become-very-slow-using-sqlcl/`

Der Oracle JDBC Driver benötigt eine Zufallszahl, um die verbindungsdaten zu verschlüsseln. Die übliche Art, wie diese mit /dev/random generiert werden, kann stocken.

Abhilfe: Aufruf von java mit:

`java -Djava.security.egd=file:///dev/urandom` *java-file argumente*

11.4 SQL-Datenbank-Zugriff in Java Stored Procedures

- Java Stored Procedures: JDBC mit dem serverseitigen JDBC-Treiber von Oracle (`jdbc:default:connection:`).
- User/Password nicht angeben, da es bereits in der DB abläuft:

```
import java.sql.*;
public class GetCountryData{
 public static void getPop (String code) throws SQLException {
  String sql = "SELECT name,population FROM country WHERE code = ?";
  try {
    Connection conn = DriverManager.getConnection("jdbc:default:connection:");
    PreparedStatement pstmt = conn.prepareStatement(sql);
    pstmt.setString(1, code);
    ResultSet rset = pstmt.executeQuery();
    if (rset.next()) System.out.println(rset.getString(2));
    conn.close();
  }
  catch (SQLException e) {
    System.err.println(e.getMessage());
  }}}
```

[Filename: Java/GetCountryData.java]

JAVA-KLASSE IN PL/SQL-PROZEDUR EINBINDEN

Laden in die Datenbank:

```
loadjava -r GetCountryData.java
```

Definition und Ausführung des Wrappers in der DB:

```
CREATE OR REPLACE PROCEDURE getPopulation (code IN VARCHAR2) AS
  LANGUAGE JAVA
  NAME 'GetCountryData.getPop(java.lang.String)';
/
```

[Filename: Java/getCountryData.sql]

... Output aktivieren:

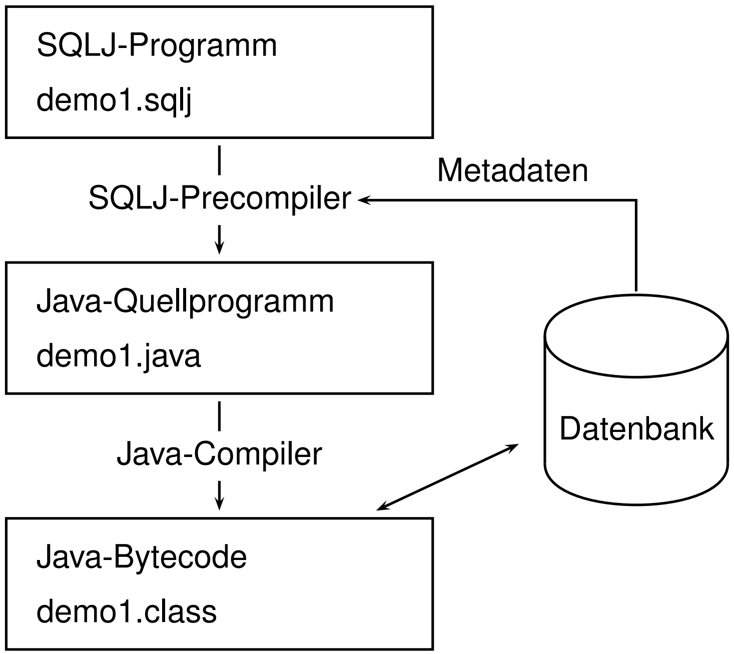
```
SET SERVEROUTPUT ON;
CALL dbms_java.set_output(2000);

EXEC getPopulation('D');
80219712
```

11.5 SQLJ

Realisierung des “Embedded SQL”-Konzeptes für Java:

- Standardisierte Spracherweiterung,
- Eingebettete SQLJ-Aufrufe werden vom Precompiler in pures Java übersetzt und dabei auf JDBC-Aufrufe abgebildet.
- **Oracle**: `sqlj` enthält den Precompiler und Compiler. Der Aufruf von `sqlj demo1.sqlj` erzeugt `demo1.java` und `demo1.class`.
- die Quelldatei muss die Endung `.sqlj` haben.
- Wenn man `demo1.java` anschaut, findet man die Umsetzung via JDBC.



Aktueller Stand (2017)

- SQLJ wird als “deprecated” (veraltet) bezeichnet.
- sqlj ruft translator.jar auf (von Oracle-Webseiten).
- Mit Java 8 zur Zeit (Juni 2017) nicht lauffähig
  (Klasse sun/io/CharToByteConverter wird nicht gefunden, existiert in Java 8 nicht mehr)
- aktuelle Installation im CIP-Pool: im sqlj-Skript wird der Classpath auf Java 7 gesetzt.

... interne Java-Version bei Oracle

(Stand Juni 2017, Oracle 12c)

```
SELECT dbms_java.get_ojvm_property(PROPSTRING
   =>'java.version')
FROM dual;

DBMS_JAVA.GET_OJVM_PROPERTY(PROPSTRING=>'JAVA.VERSION')
--------------------------------------------------------------------------------
1.6.0_151
```

- 12c: JDK 7 wird auch unterstützt, aber JDK 6 ist noch Default.

ANWEISUNGEN IN SQLJ

- Anfragen:

```
#sql anIterator
  = {SELECT name, population FROM country};
```

wobei anIterator ein (auch per SQLJ) geeignet definierter Iterator ist.

- DML und DDL:

```
#sql{<statement>};
```

- Prozeduraufrufe:

```
#sql{CALL <proc_name>[(<parameter-list>)]};
```

- Funktionsaufrufe:

```
#sql <variable>=
  {VALUES(<func_name>[(<parameter-list>)])};
```

- Aufruf unbenannter Blöcke:

```
#sql {BEGIN ... END};
```

Verbindungsaufbau zu Oracle

Ausführliche Variante

```
import java.sql.*;
import oracle.sqlj.runtime.Oracle;
//--------------
import sqlj.runtime.*;
import sqlj.runtime.ref.DefaultContext;
  :
String url = "jdbc:oracle:thin:@//xxx.xxx.xxx.xxx:1521/dbis.informatik.uni-goettingen.
String user = "...";
String passwd = "...";
DriverManager.registerDriver (new oracle.jdbc.driver.OracleDriver());
Connection con = DriverManager.getConnection(url,user,passwd);
DefaultContext ctx = new DefaultContext(con);
DefaultContext.setDefaultContext(ctx);
Oracle.connect(url, user, passwd);
//--------------
```

Verbindungsaufbau zu Oracle: Kompaktere Variante

- verwendet `conn.props`

```
import java.sql.*;
import oracle.sqlj.runtime.Oracle;
  :
Oracle.connect(<JavaClass>.class, "conn.props");
  :
```

- `<JavaClass>.class` ist eine Klasse, die im Dateisystem/jar-Archiv im selben Verzeichnis wie `conn.props` liegt (der Name dieser Klasse dient nur dazu, `conn.props` zu finden!).

HOSTVARIABLEN

- Verwendung von Variablen einer Host-Sprache (hier Java) in SQL-Statements
- Dient dem Datenaustausch zwischen Datenbank und Anwendungsprogramm
- in SQLJ-Statements wird Hostvariablen ein Doppelpunkt (":") vorangestellt
- Datentypen der Datenbank- und Programmiersprache müssen kompatibel sein (siehe JDBC)

In Host-Variablen schreiben:

```
int countries;
#sql{SELECT COUNT(*) INTO :countries FROM country};
```

Aus Host-Variablen lesen:

```
int population = 75000000;
#sql{UPDATE country SET population = :population WHERE code='D'};
```

ITERATOREN

- Allgemein: Design-Pattern, sequenzieller Zugriff auf alle Objekte, die in einem Container enthalten sind
- Hier: Iteratoren bilden das Cursor-Konzept auf SQLJ ab.
- Iteratoren mit benannten Spalten:
  - Spaltenzugriff über Spaltennamen
    `<var>:= <iterator>.name`$_i$`()`
  - Weiterschaltung mit `<iterator>.next()`
- Positionsiteratoren:
  - Spaltenzugriff über Positionen,
  - dabei Weiterschaltung
  - im embbedded-SQL-escape
    `#sql{FETCH :<iterator> INTO :<var>`$_1$`, ... , :<var>`$_n$`}`

  (Syntax im Stil des Datenbankzugriffs bei Netzwerkdatenbanken im CODASYL-Standard (1964))

Iteratoren mit benannten Spalten

Hierbei erhalten die Attribute des Iterators Namen (“Schema”):

```
import java.sql.*;
import oracle.sqlj.runtime.Oracle;
class sqljNamedIteratorExample {
 public static void main (String args []){
  try { // Datenbank-Verbindung aufbauen
   Oracle.connect(sqljNamedIteratorExample.class, "conn.props");
   // Deklaration des Iterators mit Spaltennamen und Typen
   #sql iterator CountryIter(String name, int population);
   // Iteratorinstanz definieren
   CountryIter cIter;
   // Initialisieren des Iterators mit der SQL-Anweisung
   #sql cIter = {SELECT name, population FROM country};
   // Abarbeitung der Ergebnismenge durch Iteration
   while (cIter.next()) {
      System.out.println(cIter.name() + " has " +
                cIter.population() + " inhabitants."); }
   cIter.close();
  }
  catch (SQLException e) { System.err.println(e.getMessage()); } }}
```

[Filename: Java/sqljNamedIteratorExample.sqlj]

Positionsiteratoren

```
import java.sql.*;  import oracle.sqlj.runtime.Oracle;
class sqljPosIteratorExample {
 public static void main (String args []){
  try { // Datenbank-Verbindung aufbauen
   Oracle.connect(sqljPosIteratorExample.class, "conn.props");
   // Deklaration des Iterators nur mit Typen
   #sql iterator CountryPosIterator(String, int);
   // Hilfsvariablen der Hostsprache
   String name = "";
   int pop = 0;
   CountryPosIterator cIter;   // Iteratorinstanz definieren
   // Initialisieren des Iterators mit der SQL-Anweisung
   #sql cIter = {SELECT name, population FROM country};
   // Abarbeitung der Ergebnismenge durch Iteration
   while (true) { // hole naechsten Datensatz
      #sql{FETCH :cIter INTO :name,:pop};
      // Ende des Iterators erreicht?
      if(cIter.endFetch()) break;
      System.out.println(name + " has " + pop + " inhabitants.");   }
   cIter.close();
   } catch (SQLException e) { System.err.println(e.getMessage()); } } }
```

[Filename: Java/sqljPosIteratorExample.sqlj]

Vergleich: JDBC und SQLJ

JDBC

- Call-Level-Schnittstelle
- Dynamisches SQL
- Fehlererkennung erst zur Laufzeit
- Hohe Flexibilität

SQLJ

- Embedded SQL
- Statisches SQL
- Fehlererkennung bereits zur Übersetzungszeit
- Kompakte Syntax

```
int countries;
Statement stmt = con.createStatement();
String query = "SELECT COUNT(*) FROM country";
ResultSet rset = stmt.executeQuery(query);
rset.next();
countries = rset.getInt(1);
```

```
int countries;
#sql{SELECT COUNT(*) INTO :countries FROM country};
```

11.6 Weitere SQL/Oracle-Werkzeuge

- seit ORACLE8i (1999; i= internet)
  Mit eingebauter Java Virtual Machine, Zugriff auf das Filesystem,
  Oracle-Web Server/Internet Application Server (seit 9i):
  HTML-Seiten werden abhängig vom Datenbankinhalt erstellt.
- mit den Paketen IAS, Internet File System Server wachsen Datenbank und Betriebssystem zunehmend zusammen.
- seit ORACLE9i: Integration aus der XML-Welt (*XMLType*): XPath, XSLT, DOM, XML Schema.
  ... siehe weitere Folien.
- ORACLE10g: grid computing
  Oracle Rules Manager für Aktive Ereignis-basierte Regeln

ENTWICKLUNGSLINIE ORACLE

- 1977: Gründung durch Larry Ellison, Bob Miner, Ed Oates und Bruce Scott
- 1979: erstes Produkt
- 1992: Oracle 7
- letzte 7er (1997): 7.3.4: erste JDBC-Version
- 1997/1998: Oracle 8 (bis 8.0.4): Objekttypen, Nested Tables
- 3.1999: Oracle 8i/8.1.5 (i = Internet); JVM, Java Stored Procedures & Member Methods, SQLJ
- 2.2001: Oracle 8.1.6: ein bisschen XML-Support (als Java-Tools)
- 6.2001: Oracle 9i: Java-Klassen als Object Types, Vererbung
- 5.2002: 9i-R2/9.2.0: verbesserter XML-Support (XMLType)
- 2003: Oracle 10g (g = Grid); Recycle Bin, XPath, XQuery
- 2007: 11g
- 2013: 12c: VARCHAR2 bis zu 32767 Bytes; Container-DB mit bis zu 252 Pluggable-DBs

Kapitel 12
SQL und XML

12.1 XML: “Extensible Markup Language”

... mehr als nur “Language”: Datenmodell, viele Sprachen

- Instantiierung von SGML (vgl. HTML)
- mit anwendungsspezifischen Tags

⇒ wer HTML kennt, weiß, wie XML “aussieht”.

- Baumstruktur, rekursiver Aufbau
- Elemente (Name, Attribute und Inhalt)
- Navigation im Baum
- vgl. Pfadausdrücke in Java, OQL, SQL; (z.B. x.coordinates.latitude)
- Pfadausdrücke in Unix: (z.B. /home/may/teaching/dbp/folien.tex)

⇒ Adressierungssprache “XPath”

XML: Beispiel

```
<country id="D" capital="cty-Germany-Berlin">
    <name>Germany</name>
    <total_area>356910</total_area>
    <population year="2011">80219695</population>
    <encompassed continent="europe">100</encompassed>
    <ethnicgroup name="German">85</ethnicgroup>
    <ethnicgroup name="Polish">2.5</ethnicgroup>
    <religion name="Roman Catholic">28.5</religion>
    <religion name="Protestant">27.5</religion>
    <language name="German">95</language>
    <border country="F">451</border>
    <border country="A">784</border>
    <border country="CZ">646</border>
    <province id="prov-Germany-Baden-Wuerttemberg">
      <name>Baden Württemberg</name>
      <area>35742</area>
      <population year="2011">10486660</population>
      <city id="cty-Germany-Stuttgart">
        <name>Stuttgart</name>
          :
```

```
          <latitude>48.7</latitude>
          <longitude>9.1</longitude>
          <population year="2011">585890</population>
        </city>
        <city id="cty-Germany-Mannheim">
          <name>Mannheim</name>
          :
        </city>
        :
      </province>
      <province id="prov-Germany-Berlin">
        <name>Berlin</name>
        <area>889</area>
        <population year="2011">3292365</population>
        <city id="cty-Germany-Berlin">
          <name>Berlin</name>
          <latitude>52.45</latitude>
          <longitude>13.3</longitude>
          <population year="2011">3292365</population>
        </city>
      </province>
      :
    </country>
```

XML

(Siehe Vorlesung “Semistrukturierte Daten und XML”)

- Verwendung:
  - Dokumente
  - Datenaustausch
  - Datenspeicherung
- sehr flexibles “Datenmodell”: DOM-API
  rekursiv definierte Baumstruktur aus
  - Elementen,
  - Attributen und
  - Textknoten.
- Schema: DTD (Document Type Description), XML Schema
- Erweiterungen: XPath, XPointer, XLink
- Anfragesprache: XQuery
- Transformationssprache: XSL/XSLT
- Als Repräsentationssprache für viele Anwendungen genutzt

12.2 Der SQL/XML bzw. SQLX Standard – Kombination relationaler Daten und XML

- Abbildung von relationalen Daten nach XML
- Speicherung von XML-Daten in RDBMS
- Entwurf eines ISO-Standards seit 2003: `www.sqlx.org`
- SQL-Objektdatentyp "XMLType"
  - mit entsprechenden Konstruktoren für XML-Strukturen,
  - und Zugriffsmethoden (basierend auf den Standards der XML-Welt),
  - benutzbar von SQL und innerhalb von PL/SQL.
- zum Teil noch unvollständig und überraschend ...
- Folien: siehe Vorlesung "Semistructured Data and XML"