TEIL III: Erweiterungen

Teil I: Grundlagen

Teil II: Diverses

Teil III: Prozedurale Konzepte, OO, Einbettung

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

SITUATION

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

ERWEITERUNGEN

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

Kapitel 8
Prozedurale Erweiterungen: PL/SQL

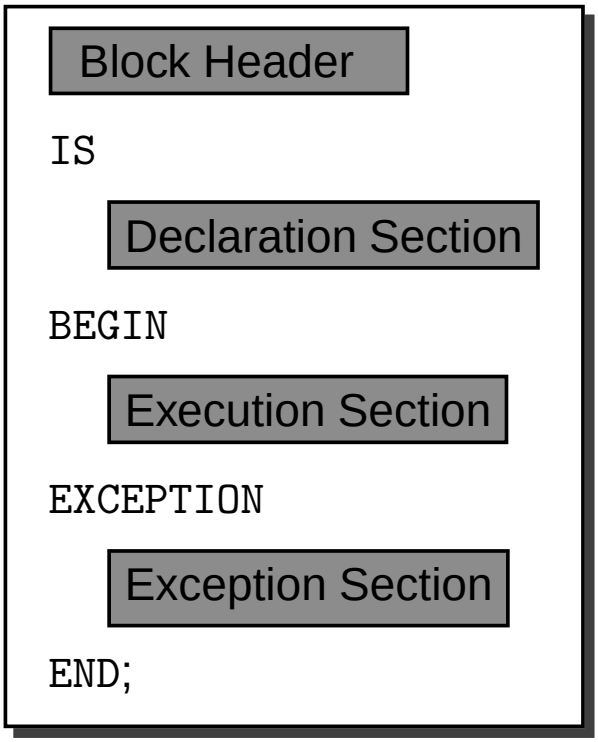
- Erweiterung von SQL um prozedurale Elemente *innerhalb* der SQL-Umgebung, *PL/SQL* (*Procedural language extensions to SQL*).
- “Stored Procedures/Functions” innerhalb der DB
- direkter Zugriff auf Datenbankinhalt
- Vorteile von PL/SQL: Bessere Integration der prozeduralen Elemente in die Datenbank;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

Weitere Nutzung

- Programmierung von Objektmethoden (seit Oracle 8/1997)

8.1 Prozeduren, Funktionen und Kontrollstrukturen in PL/SQL

Blockstruktur von PL/SQL



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

EINFACHE, ANONYME BLÖCKE

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

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 können mit `DROP PROCEDURE/FUNCTION <name>` gelöscht werden.
- 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(30);
```

anchored TYPDEKLARATION

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

```
<variable> <variable'>%TYPE
  [NOT NULL] [DEFAULT <value>];
```

oder

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

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

ZUWEISUNG AN VARIABLEN

- “klassisch” innerhalb des Programms:

```
a := b;
```

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

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

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

[Filename: PLSQL/simple.sql]

PL/SQL-Datentypen: Records

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

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

the_city city_type;
```

anchored Typdeklaration für Records

Records mit Tabellenzeilen-Typ deklarieren: %ROWTYPE:

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

Äquivalent zu oben:

```
the_city city%ROWTYPE;
```

Zuweisung an Records

- Aggregierte Zuweisung: zwei Variablen desselben Record-Typs:

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

- Feldzuweisung: ein Feld wird einzeln zugewiesen:

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

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

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

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

[Filename: PLSQL/simple2.sql]

Vergleich von Records

Beim Vergleich von Records muss jedes Feld einzeln verglichen werden.

PL/SQL-Datentypen: PL/SQL Tables

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

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

<var> <type>;
```

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

plz_table plz_table_type;
```

- Adressierung: <var>(1)

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

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

Tabellen können auch als Ganzes zugewiesen werden

```
andere_table := plz_table;
```

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

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

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

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

SQL-Statements in PL/SQL

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

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

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

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

KONTROLLSTRUKTUREN

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

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

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

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

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

GESCHACHTELTE BLÖCKE

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

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

8.2 Cursore/Iteratoren zur Verarbeitung von Ergebnismengen

- Datenbankanfragen: mengenorientiert
- Programmiersprache: variablenbasiert

Design Patterns: Kollektionen und Iteratoren

(vgl. Informatik I)

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

... Iteratoren werden im Weiteren immer wieder verwendet.

CURSORBASIERTER DATENBANKZUGRIFF

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

Cursordeklaration in der *Declaration Section*:

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

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

Beispiel:

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

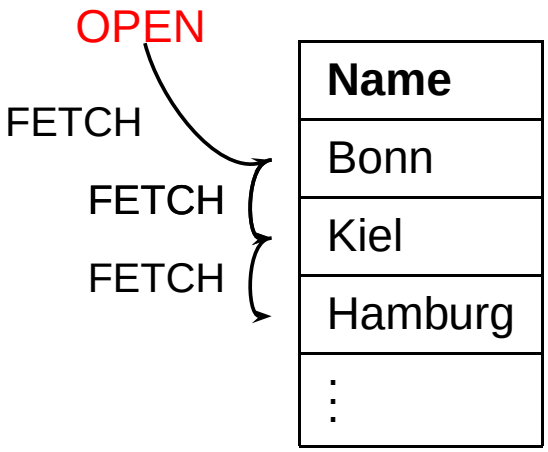
```
DECLARE
CURSOR cities_in (the_country Country.Code%TYPE)
IS SELECT Name
   FROM City
   WHERE Country=the_country;
```

Cursore: Grundprinzip

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

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

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



Cursore: Verwendung

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

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

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

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

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

[Filename: PLSQL/cursor1.sql]

Cursore: Attribute

Kontrolle über die Verarbeitung eines Cursors:

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

Cursore: Attribute

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

[Filename: PLSQL/cursor-attrs.sql]

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

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

Cursore: Hinweis

nicht möglich:

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

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

CURSOR FOR LOOP

Spezielle Schleife zur Iteration über den Inhalt eines Cursors:

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

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

Cursor FOR LOOP: Beispiel

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

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

[Filename: PLSQL/cursor-loop1.sql]

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

EINGEBETTETER CURSOR FOR LOOP

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

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

[Filename: PLSQL/cursor-loop2.sql]

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

SCHREIBZUGRIFF VIA CURSOR

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

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

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

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

DDL-Kommandos in PL/SQL

DDL-Statements werden in PL/SQL nicht direkt unterstützt:

- `EXECUTE IMMEDIATE <string>`
  <string> kann dabei eine Konstante sein, oder kann dynamisch zusammengesetzt werden

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

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;
/
```

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

DYNAMIC SQL MIT EXECUTE IMMEDIATE

- Werte für Variablen einsetzen: USING :n
  (ebenso bei Aufruf von Prozeduren/Funktionen)

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

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

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

8.3 Zugriffsrechte auf PL/SQL-Datenbankobjekte

Benutzung von Funktionen/Prozeduren:

- Benutzungsrechte vergeben:
  `GRANT EXECUTE ON <procedure/function> TO <user>;`
- Prozeduren und Funktionen werden jeweils mit den Zugriffsrechten des *Besitzers* ausgeführt.
- nach

  `GRANT EXECUTE ON <procedure/function> TO <user>;`

  kann dieser User die Prozedur/Funktion auch dann aufrufen, wenn er kein Zugriffsrecht auf die dabei benutzten Tabellen hat.
- Möglichkeit, Zugriffsberechtigungen strenger zu formulieren als mit `GRANT ... ON <table> TO ...`: Zugriff nur in einem ganz speziellen, durch die Prozedur oder Funktion gegebenen Kontext.

8.4 Geschachtelte Tabellen unter PL/SQL

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

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

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

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

Geschachtelte Tabellen unter PL/SQL: Beispiel

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

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

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

(RE)AKTIVES VERHALTEN

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

8.5 Trigger

Einschub: Integritätsbedingungen

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

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

Assertions

- Bedingungen, die den gesamten DB-Zustand betreffen.

  `CREATE ASSERTION <name> CHECK (<bedingung>)`

- Diese werden allerdings von Oracle bisher nicht unterstützt.

⇒ Also muss man sich etwas anderes überlegen.

TRIGGER

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

TRIGGER

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

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

TRIGGER: BEISPIEL

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

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

[Filename: PLSQL/changecode.sql]

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

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

TRIGGER: BEISPIEL

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

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

[Filename: PLSQL/newcountry.sql]

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

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

TRIGGER: MUTATING TABLES

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

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

Nachteil: Oft ein zu strenges Kriterium.

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

INSTEAD OF-TRIGGER

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

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

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

VIEW UPDATES UND INSTEAD OF-TRIGGER

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

[Filename: PLSQL/allcountry-view.sql]

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

[Filename: PLSQL/insert-allcountry.sql]

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

VIEW UPDATES UND INSTEAD OF-TRIGGER

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

[Filename: PLSQL/instead-of.sql]

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

FEHLERBEHANDLUNG

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

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

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

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

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

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

ABLAUF

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

TRIGGER/FEHLERBEHANDLUNG: BEISPIEL

Nachmittags dürfen keine Städte gelöscht werden:

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

[Filename: PLSQL/trigger-nachmittag.sql]

BEISPIEL

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

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

1 Zeile wurde erstellt.

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

Es wurden keine Zeilen ausgewaehlt.
```

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

8.6 Weitere PL/SQL-Features

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

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

Kapitel 9
Objekt-Relationale Datenbanksysteme

Integration von relationalen Konzepten und Objektorientierung:

- Komplexe Datentypen: Erweiterung des Domain-Konzepts von SQL-2 (vgl. `DATE`, Geo-Koordinaten)
- Abstrakte Datentypen (“Objekttypen”):
  - Unterscheidung zwischen dem *Zustand* und *Verhalten* eines *Objektes* (Kapselung interner Funktionalität).
  - Im Gegensatz zu einem *Tupel* besitzt ein Objekt
    - Attribute (beschreiben seinen Zustand),
    - Methoden – Abfragen und Ändern des Zustandes: *Prozeduren* und *Funktionen*
      (Oracle 8: PL/SQL, Oracle 8i: auch Java, siehe Folie 285)
    - `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 334).

STUFEN DER OBJEKTORIENTIERUNG

"Konservative" objektrelationale Erweiterungen (seit Oracle 8)

(siehe Folie 208)

- 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 233)

- 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 285 und 289)

OBJEKTTYPDEKLARATION

```
CREATE [OR REPLACE] TYPE <type> AS OBJECT
  (<attr> <datatype>,
   <attr> <datatype>,
   ⋮
   MEMBER FUNCTION <func-name> [(<parameter-list>)]
      RETURN <datatype>,
   ⋮
   MEMBER PROCEDURE <proc-name> [(<parameter-list>)],
   ⋮
   [ MAP MEMBER FUNCTION <func-name>
         RETURN <datatype>, |
     ORDER MEMBER FUNCTION <func-name>(<var> <type>)
         RETURN <datatype>, ]
   [ <pragma-declaration-list>]
  );
/
```

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

Beispiel: Geo-Koordinaten

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

```
CREATE OR REPLACE TYPE GeoCoord AS OBJECT
  (Longitude NUMBER,
   Latitude NUMBER,
   MEMBER FUNCTION
      Distance (other IN GeoCoord)
      RETURN NUMBER,
   MAP MEMBER FUNCTION
      Distance_Greenwich RETURN NUMBER,
   PRAGMA RESTRICT_REFERENCES
      (Distance, WNPS, WNDS, RNPS, RNDS),
   PRAGMA RESTRICT_REFERENCES
      (Distance_Greenwich, WNPS, WNDS, RNPS, RNDS)
  );
/
```

[Filename: ObjRel/geocoord-type.sql]

TYPE BODY

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

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

[Filename: ObjRel/geocoord-body.sql]

Erzeugung von Objekten

- Konstruktormethode:

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

  Also kein `NEW`, sondern nur einfach

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

Methodenaufruf

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

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

Beispiel

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

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

9.1.2 Veraltet (bis einschl. Version 8i notwendig)

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

```
CREATE [OR REPLACE] TYPE <type> AS OBJECT
  (<attr> <datatype>,
   <attr> <datatype>,
   ⋮
   MEMBER FUNCTION <func-name> [(<parameter-list>)]
      RETURN <datatype>,
   ⋮
   MEMBER PROCEDURE <proc-name> [(<parameter-list>)],
   ⋮
   [ MAP MEMBER FUNCTION <func-name>
         RETURN <datatype>, |
     ORDER MEMBER FUNCTION <func-name>(<var> <type>)
         RETURN <datatype>, ]
   [ <pragma-declaration-list>]
  );
/
```

PRAGMA-KLAUSELN:

Read/Write-Zugriffscharakteristik

<pragma-declaration-list>:

für jede Methode eine PRAGMA-Klausel

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

<feature-list>:

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

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

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

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

```
PRAGMA RESTRICT_REFERENCES
  (<function-name>, WNDS, WNPS, RNPS, RNDS)
```

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

Beispiel: Geo-Koordinaten mit PRAGMA-Klausel

```
CREATE OR REPLACE TYPE GeoCoord AS OBJECT
  (Longitude NUMBER,
   Latitude NUMBER,
   MEMBER FUNCTION
      Distance (other IN GeoCoord)
      RETURN NUMBER,
   MAP MEMBER FUNCTION
      Distance_Greenwich RETURN NUMBER,
   PRAGMA RESTRICT_REFERENCES
      (Distance, WNPS, WNDS, RNPS, RNDS),
   PRAGMA RESTRICT_REFERENCES
      (Distance_Greenwich, WNPS, WNDS, RNPS, RNDS)
  );
/
```

9.1.3 Verwendung von Objekttypen

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

SPALTENOBJEKTE

- Attribut eines Tupels oder eines Objekts ist objektwertig,

```
CREATE TABLE Mountain
  (Name VARCHAR2(20)
      CONSTRAINT MountainKey PRIMARY KEY,
   Height NUMBER
      CONSTRAINT MountainHeight CHECK (Height >= 0),
   Coordinates GeoCoord CONSTRAINT MountainCoord
      CHECK ((Coordinates.Longitude > -180) AND
             (Coordinates.Longitude <= 180) AND
             (Coordinates.Latitude >= -90) AND
             (Coordinates.Latitude <= 90)));
```

[Filename: ObjRel/mountain-table.sql]

- Constraints werden wie immer bei der Tabellendefinition angegeben.

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

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

ZEILENOBJEKTE

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

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

mit `<constraint-list>` wie bisher:

- attributbezogene Bedingungen entsprechen den Spaltenbedingungen:

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

- Tabellenbedingungen: Syntax wie bei Tupeltabellen.

ZEILENOBJEKTE

Beispiel: *City_Type*

```
CREATE OR REPLACE TYPE City_Type AS OBJECT
  (Name VARCHAR2(35),
   Province VARCHAR2(32),
   Country VARCHAR2(4),
   Population NUMBER,
   Coordinates GeoCoord,
   MEMBER FUNCTION Distance (other IN City_Type)
      RETURN NUMBER,
   MEMBER FUNCTION NoOfOrganizations
      RETURN NUMBER);
/
```

[Filename: ObjRel/citytype.sql]

ZEILENOBJEKTE

Beispiel: *City_Type*

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

[Filename: ObjRel/citytypebody.sql]

OBJEKTTABELLEN: ZEILENOBJEKTE

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

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

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

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

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

VERWENDUNG VON OBJEKTTABELLEN

Auslesen und Ändern von Attributen wie bekannt

- Auslesen:

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

- Ändern:

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

Methodenaufrufe wie vermutet

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

VERWENDUNG VON OBJEKTTABELLEN

... Auslesen von Objekten als Objekte:

Das folgende geht so *nicht*:

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

Die VALUE()-Funktion

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

selektiert ein Objekt als Objekt:

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

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

VERWENDUNG VON OBJEKTEN: VALUE

- Objekte auf Gleichheit testen
- Objekt als Argument einer Methode

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

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

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

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

9.1.4 ORDER- und MAP-Methoden

- Objekttypen besitzen im Gegensatz zu den Datentypen NUMBER und VARCHAR keine inhärente Ordnung.
- Ordnung auf Objekten eines Typs kann über dessen funktionale Methoden definiert werden.
- ORACLE 8: für jeden Objekttyp eine MAP FUNCTION oder ORDER FUNCTION.

MAP-Funktion: (Betragsfunktion)

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

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

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

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

MAP-METHODEN: BEISPIEL

MAP-Methode auf *GeoCoord*:

```
CREATE OR REPLACE TYPE BODY GeoCoord
AS
 :
MAP MEMBER FUNCTION Distance_Greenwich
  RETURN NUMBER
  IS
  BEGIN
   RETURN SELF.Distance(GeoCoord(0, 51));
  END;
END;
/
```

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

[Filename: ObjRel/orderby.sql]

ORDER-METHODEN

- Vergleich von SELF mit einem anderen Objekt desselben Typs, das formal als Parameter angegeben wird.
- Ergebnis: NUMBER
  - $x < 0$ falls SELF < Parameter,
  - 0 (Gleichheit), oder
  - $x > 0$ falls SELF > Parameter.
- Wird ORDER BY angegeben, werden die Ausgabeobjekte paarweise verglichen und entsprechend der ORDER-Methode geordnet.
- Ein Beispiel hierfür ist etwa die Erstellung der Fussball-Bundesligatabelle: Ein Verein wird vor einem anderen plaziert, wenn er mehr Punkte hat. Bei Punktgleichheit entscheidet die Tordifferenz. Ist auch diese dieselbe, so entscheidet die Anzahl der geschossenen Tore (vgl. Aufgabe).

9.1.5 Objektrelationale Modellierung: Zusammenfassung

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

BEISPIEL/AUFGABE

(wird auf Folie 243 analog ausprogrammiert)

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

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

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

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

9.2 Objektorientierte Modellierung

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

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

Objektorientierte Modellierung

Geht über die Nutzung als “Datentypen” hinaus ...

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

OBJEKTREFERENZEN

- Weiterer Datentyp für Attribute: Referenzen auf Objekte

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

- *Objekttyp* als Ziel der Referenz.
- nur Objekte, die eine OID besitzen – also Zeilenobjekte einer Objekttabelle – können referenziert werden.
- problemlose Integration referentieller Integritätsbedingungen von Objekttabellen zu bestehenden relationalen Tabellen.
- `PRIMARY KEY`s dürfen keine `REF`-Attribute umfassen.
- Objekttyp kann in verschiedenen Tabellen vorkommen: Einschränkung auf eine bestimmte Tabelle bei der Deklaration der entsprechenden Tabelle als Spalten- oder Tabellenconstraints mit `SCOPE`:
  - als Spaltenconstraint (nur bei Tupeltabellen):

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

  - als Tabellenconstraint:

    ```
    SCOPE FOR (<ref-attr>) IS <object-table>
    ```

- Erzeugen einer Referenz (Selektieren einer OID):

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

Beispiel: Objekttyp *Organization*

```
CREATE TYPE Member_Type AS OBJECT
 (Country VARCHAR2(4),
  Type VARCHAR2(30));
/
CREATE TYPE Member_List_Type AS
 TABLE OF Member_Type;
/
CREATE OR REPLACE TYPE Organization_Type AS OBJECT
  (Name VARCHAR2(80),
   Abbrev VARCHAR2(12),
   Members Member_List_Type,
   Established DATE,
   has_hq_in REF City_Type,
   MEMBER FUNCTION is_member (the_country IN VARCHAR2)
   -- EU.is_member('SLO') = 'membership applicant'
      RETURN VARCHAR2,
   MEMBER FUNCTION people RETURN NUMBER,
   MEMBER FUNCTION number_of_members RETURN NUMBER,
   MEMBER PROCEDURE add_member
      (the_country IN VARCHAR2, the_type IN VARCHAR2));
/
```

[Filename: Obj/org-type.sql]

Beispiel: Objekttyp *Organization*

Tabellendefinition:

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

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

Einfügen unter Verwendung des Objektkonstruktors:

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

Setzen des Referenzattributes *has_hq_in*:

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

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

SELEKTION VON OBJEKTATTRIBUTEN

- Wertattribute

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

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

- Referenzattribute:

  SELECT <ref-attr-name>

  liefert OID:

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

Name	Abbrev	has_hq_in
European Community	EU	<oid>

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

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

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

VERWENDUNG VON REFERENZATTRIBUTEN

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

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

  adressiert ("*navigierender Zugriff*").
- Aliasing mit einer Variablen um den Pfadausdruck eindeutig zu machen:

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

Abbrev	has_hq_in.Name
EU	Brussels

Die Funktionen VALUE, REF, DEREF

Mit REF und DEREF lässt sich VALUE ersetzen:

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

und

```
SELECT DEREF(REF(cty)) FROM City_ObjTab cty;
```

sind äquivalent.

ZYKLISCHE REFERENZEN

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

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

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

- wird später durch eine komplette Typdeklaration ergänzt.

UNVOLLSTÄNDIGE DATENTYPEN

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

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

- Die Nutzung in Referenzen ist damit erlaubt:

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

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

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

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

CREATE OR REPLACE TYPE Country_Type_2 AS OBJECT
  (Name VARCHAR2(32),
   Code VARCHAR2(4),
   Capital City_Type);
/ -- Capital waere ein Spaltenobjekt
```

Zyklische Referenzen: Beispiel

```
CREATE OR REPLACE TYPE City_Type
/

CREATE OR REPLACE TYPE Country_Type AS OBJECT
  (Name VARCHAR2(32),
   Code VARCHAR2(4),
   Capital REF City_Type,
   Area NUMBER,
   Population NUMBER);
/

CREATE OR REPLACE TYPE Province_Type AS OBJECT
  (Name VARCHAR2(32),
   Country REF Country_Type,
   Capital REF City_Type,
   Area NUMBER,
   Population NUMBER);
/

CREATE OR REPLACE TYPE City_Type AS OBJECT
  (Name VARCHAR2(35),
   Province REF Province_Type,
   Country REF Country_Type,
   Population NUMBER,
   Coordinates GeoCoord);
/
```

OBJEKTORIENTIERUNG: MODELLIERUNGSASPEKTE

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

Beispiel/Aufgabe

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

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

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

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 285).

Member-Methods: Beispiel

```
CREATE OR REPLACE TYPE BODY Organization_Type IS

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

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

Member-Methods: Beispiel (Forts.)

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

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

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

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

METHODENAUFRUFE

Funktionen

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

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

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

Prozeduren

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

$\Rightarrow$ freie Prozeduren in PL/SQL, um `MEMBER PROCEDURES` aufzurufen

Beispiel: Freie Prozedur

```
CREATE OR REPLACE PROCEDURE make_member
  (the_org IN VARCHAR2, the_country IN VARCHAR2,
   the_type IN VARCHAR2) IS
  n NUMBER;
  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.is_member(the_country)='no' THEN
   x.add_member(the_country, the_type);
  END IF;
END;
/
```

[Filename: Obj/makemember.sql]

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

Beispiel: Füllen der Objekttabelle

Übertragung des Datenbestandes aus den relationalen Tabellen *Organization* und *is_member* in die Objekttabelle *Organization_ObjTab*:

```
INSERT INTO Organization_ObjTab
 (SELECT Organization_Type
     (Name, Abbreviation, NULL, Established, NULL)
  FROM Organization);
CREATE OR REPLACE PROCEDURE Insert_All_Members IS
BEGIN
 FOR the_membership IN
     (SELECT * FROM is_member)
 LOOP make_member(the_membership.organization,
                  the_membership.country,
                  the_membership.type);
 END LOOP;
END;
/
EXECUTE Insert_All_Members;
UPDATE Organization_ObjTab org
SET has_hq_in =
  (SELECT REF(cty)
   FROM ORCity cty, Organization old
   WHERE org.Abbrev = old.Abbreviation
   AND cty.Name = old.City
   AND cty.Province = old.Province
   AND cty.Country = old.Country);
```

[Filename: Obj/fill-organizations.sql]

Beispiel: Nutzung freier Methoden

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

[Filename: Obj/is-member.sql]

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

is_member_in('EU', 'CZ')
applicant

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

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

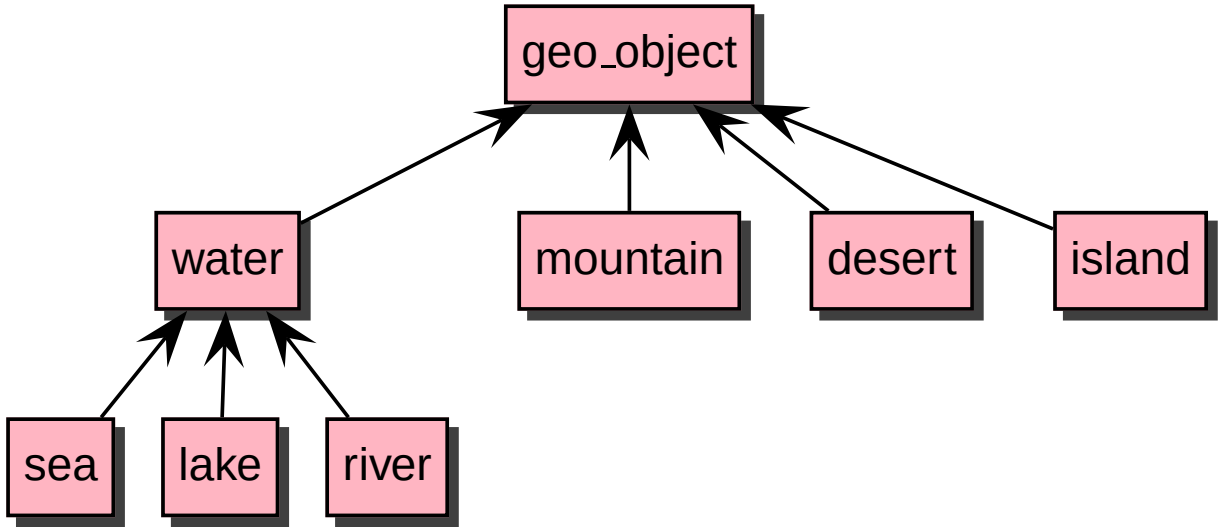
MODELLIERUNG VS. IMPLEMENTIERUNG

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

$\Rightarrow$ für den Benutzer bleiben die freien Methoden dieselben.

9.4 Klassenhierarchie und Vererbung

(Abschnitt ist optional)



KLASSENHIERARCHIE UND VERERBUNG

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

Erzeugen von Subtypen

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

```
CREATE OR REPLACE TYPE geo_object_type AS OBJECT (
   name VARCHAR2(25),
   MEMBER FUNCTION get_name RETURN VARCHAR2,
   NOT INSTANTIABLE
          MEMBER FUNCTION set_name RETURN VARCHAR2
  )
  NOT INSTANTIABLE -- DEFAULT: INSTANTIABLE
  NOT FINAL;       -- DEFAULT: FINAL
/

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

Erzeugen von Subtypen

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

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

Erzeugen von Subtypen

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

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

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

TABELLEN ÜBER ALLGEMEINEN KLASSEN

- eine Tabelle für alle geographischen Objekte

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

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.height
```

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

TYPUMWANDLUNGEN

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

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

9.5 Diverses zu Objekttypen

Änderungen an Objekttypen

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

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

```
ALTER TYPE <type>
  ADD ATTRIBUTE (<name> <datatype>),
  DROP ATTRIBUTE <name>,
  MODIFY ATTRIBUTE (<name> <datatype>),
  ADD MEMBER FUNCTION/PROCEDURE <method-spec>
      -- requires new CREATE TYPE BODY!
  DROP MEMBER FUNCTION/PROCEDURE <method-spec>
      <options>
```

Änderung von Typdefinitionen: Abhängigkeiten

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

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

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

INDEXE AUF OBJEKTATTRIBUTEN

Indexe können auch auf Objektattributen erstellt werden:

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

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

```
-- nicht erlaubt:
CREATE INDEX city_index
   ON City_ObjTab(coordinates);
```

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

```
CREATE INDEX city_index
   ON City_ObjTab(coordinates.Longitude,
                  coordinates.Latitude);
```

- Funktions-basierte Indexe:

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

  arbeiten mit vorberechneten Werten.

ZUGRIFFSRECHTE AUF OBJEKTE

Recht an Objekttypen:

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

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

REFERENTIELLE INTEGRITÄT

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

  Überprüfung durch

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

  Verwendung z.B. in einem `AFTER`-Trigger:

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

9.6 Object-Views

- maßgeschneiderte Object-Views mit sehr weitgehender Funktionalität

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

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

Effizienz + Benutzerfreundlichkeit:

Die relationale Repräsentation ist oft effizienter:

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

$\Rightarrow$ Definition eines relationalen Basisschemas mit Object-Views.

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

$\Rightarrow$ Veränderungen durch Neudefinition geeigneter Object-Views abfangen.

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

OBJECT-VIEWS

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

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

Syntax

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

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

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

OBJECT VIEWS: *Country*

```
CREATE OR REPLACE TYPE Country_Type AS OBJECT
 (Name        VARCHAR2(32),
  Code        VARCHAR2(4),
  Capital     REF City_Type,
  Area        NUMBER,
  Population NUMBER);
/
```

Sinnvollerweise würde man hier gleich auch noch Methoden definieren.

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

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

OBJECT VIEWS: WAS NICHT GEHT

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

Object View auf Basis von *Organization_ObjTab*:

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

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

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

Beide angegebenen Attribute sind auch einzeln nicht erlaubt.

9.7 Fazit

- *Objektrelationale Tabellen* (Folie 208):
  Kompatibilität mit den grundlegenden Konzepten von SQL. U.a. Fremdschlüsselbedingungen von objektrelationalen Tabellen zu relationalen Tabellen.
- *Objektorientiertes Modell* (Folie 233):
  ... etwas kompliziert zu handhaben.
- *Object/Objekt-Relationale Views* (Folie 266):
  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 334), 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)
- 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 322) und weitere.

Impedance Mismatch bei der SQL-Einbettung

- Typsysteme passen nicht zusammen
- Unterschiedliche Paradigmen:
  Mengenorientiert vs. einzelne Variablen

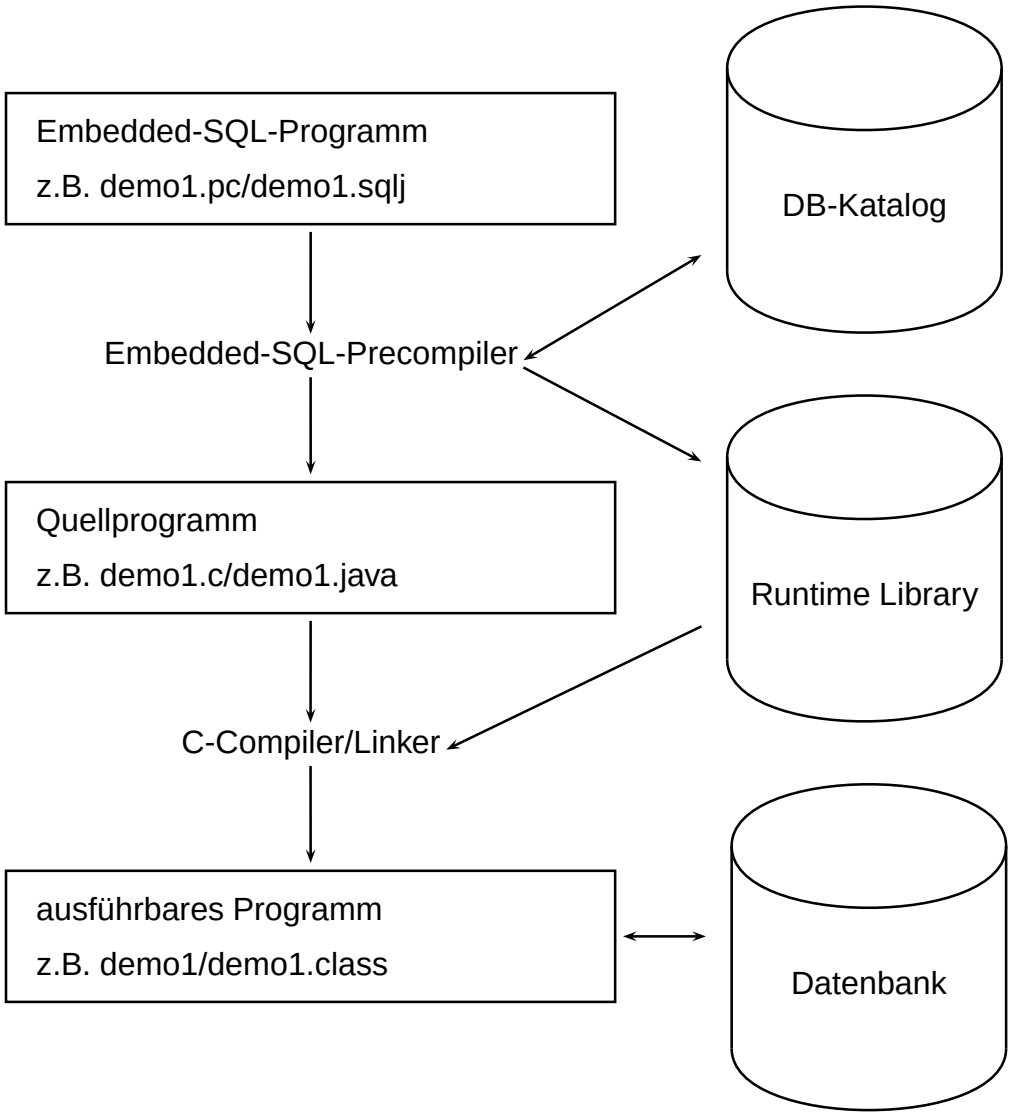
Realisierte Lösung

- 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 322): Zwischenschritt bei der Compilierung nicht sichtbar.

10.2 Embedded SQL in C [Legacy]

Hinweis: dieser Abschnitt kann ausgelassen und durch SQLJ (Folie 322) 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[25];     /* output host var */
      int cityEinw;          /* output host var */
      char* landID = "D";     /* input host var */
      short ind1, ind2;       /* indicator var */
      char* uid = "/";
   EXEC SQL END DECLARE SECTION;
   /* Verbindung zur Datenbank herstellen */
   EXEC SQL CONNECT :uid;
   /* Cursor deklarieren */
   EXEC SQL DECLARE StadtCursor CURSOR FOR
      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)
    {  /* keine NULLwerte ausgeben */
       printf("%s     %d \n", cityName, cityEinw);
    }};
   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[25][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>`

MECHANISMEN FÜR 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.
- API's: 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 285),
- Objekttypen: Java Object Types (Folie 289)
  (so kann man beliebige Datenstrukturen implementieren und anbieten → XML),
- Low-Level-Infrastruktur für Datenbankzugriff aus Java: JDBC (Folie 293),
- Embedded SQL (intern basierend auf JDBC): SQLJ (Folie 322).

11.1 Java in Stored Procedures und Member Methods

- Oracle hat (seit 8i) 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 293 und 322).
- Quelldateien (`.java`), Klassendateien (`.class`) oder Archive (`.jar`) können eingelesen werden.
- Einlesen (shell): `loadjava`
- Löschen (shell): `dropjava`
- Einbettung in Prozedur/Funktion (*Wrapper*, *call spec*) (void-Methoden als Prozeduren, non-void als Funktionen)

LADEN VON JAVA-CODE

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]

- Speichern als `Greet.java`,
- Einlesen in die Datenbank mit `loadjava`.
  Empfehlenswert ist hier ein Alias:

  `alias loadjava='loadjava -u` $uname/passwd$`'`

  dann braucht das Passwort nicht angegeben zu werden:

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

- Das Kompilieren der Klasse erfolgt in der Datenbank.
- Alternativ: Einlesen von .class-Dateien (ohne `-r`):

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

- Löschen einer Java-Klasse: analog mit `dropjava`

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!

- Um die Java-Ausgabe auch zu bekommen, 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 Abschnitt über JDBC
- Achtung: In der Namensspezifikation 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 293).

11.2 Java-Objekttypen

Man kann Java-Klassen als SQL-Typen registrieren. Die Java-Klasse muss das Interface `java.sql.SQLData` implementieren.
Methoden:

- `public String getSQLTypeName()`
  liefert den entsprechenden SQL-Datentyp zurück
- `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

```
import java.sql.*;

public class GeoCoordJ implements java.sql.SQLData {
 private double longitude, latitude;

  public String getSQLTypeName() {
    return "SCOTT.GEOCOORD";
 }

 public void readSQL(SQLInput stream, String typeName)
   throws SQLException {
     longitude = stream.readDouble();
     latitude = stream.readDouble();
 }

 public void writeSQL(SQLOutput stream)
   throws SQLException {
     stream.writeDouble(longitude);
     stream.writeDouble(latitude);
 } //... to be continued
```

- `SCOTT.GEOCOORD`: Name des SQL-Typs in Oracle
- Felder lesen/setzen in der Reihenfolge der SQL-Definition
- Lese-/Schreibmethoden: `stream.read/write<type>`

BEISPIEL CONT.: GeoCoordJ

```
//... continued
  public double distance(GeoCoordJ other) {
    return
      6370 *
      Math.acos(
         Math.cos(this.latitude/180*3.14) *
         Math.cos(other.latitude/180*3.14) *
         Math.cos(
          (this.longitude - other.longitude)
             /180*3.14
         ) +
         Math.sin(this.latitude/180*3.14) *
         Math.sin(other.latitude/180*3.14)
      );
  }
}
```

[Filename: Java/GeoCoordJ.java]

SQL-Wrapper-Type

```
CREATE OR REPLACE TYPE geocoord
AS OBJECT EXTERNAL
NAME 'GeoCoordJ'
LANGUAGE JAVA USING SQLData(
longitude number external name 'longitude',
latitude number external name 'latitude',
MEMBER FUNCTION distance (other IN GeoCoord)
  RETURN NUMBER EXTERNAL
  NAME 'distance (GeoCoordJ) return double');
/
CREATE TABLE geoTable OF geocoord;
INSERT INTO geoTable VALUES (geocoord(10,20));
INSERT INTO geoTable VALUES (geocoord(20,30));

SET SERVEROUTPUT ON
CALL dbms_java.set_output(2000);

SELECT g.distance(geocoord(0,51)) FROM geoTable g;
```

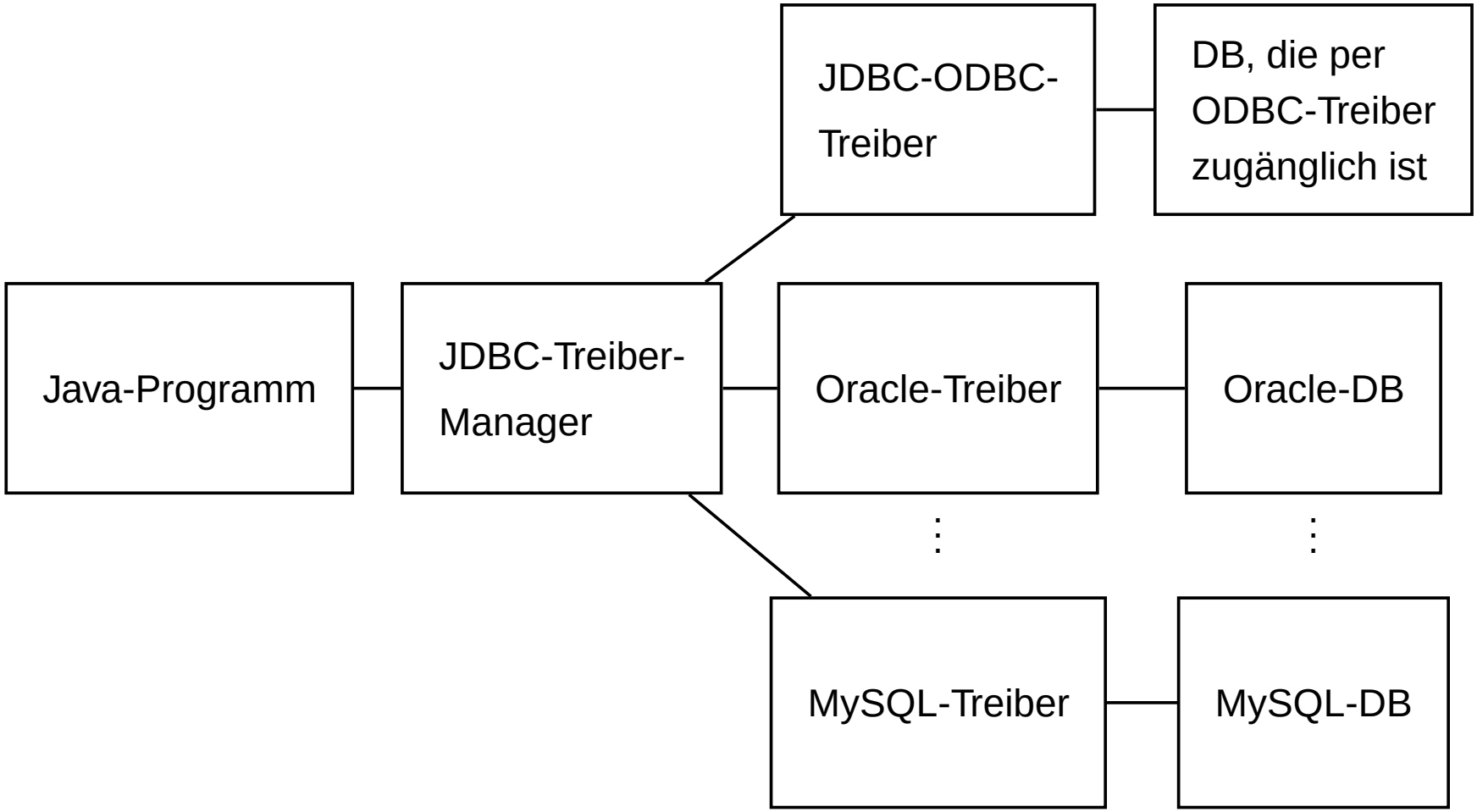
[Filename: Java/GeoCoordJ.sql]

```
  G.DISTANCE(GEOCOORD(0,51))
-------------------------
                 5530.99686
                 4708.68206
```

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.

$\Rightarrow$ Klasse `DriverManager`:

- nur `static` Methoden (operieren auf Klasse)
- Konstruktor ist `private` (keine Instanzen erzeugen)

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

  ```
  DriverManager.getConnection(<jdbc-url>, <props>);
  ```

  (Login-Daten in externer Datei, `java.util.Properties`).

- Datenbank wird eindeutig durch JDBC-URL bezeichnet

JDBC-URL:

- `jdbc:<subprotocol>:<subname>`
- `<subprotocol>`: Treiber und Zugriffsmechanismus
- `<subname>` bezeichnet Datenbank

Bei uns:

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

```
String url =
  'jdbc:oracle:thin:@xxx.xxx.xxx.xxx:1521:dbis';
```

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.
- `<statement>.execute(<string>)` – Sonstiges:
  - Aufrufe von Prozeduren/Funktionen (siehe CallableStatements)
  - Statement gibt mehr als eine Ergebnismenge zurück (Folge von Anweisungen).
    Ergebnismengen etc. sind dann nacheinander über das Statement-Objekt abfragbar (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`
`DATE`	`getDate`
`TIME`	`getTime`

  `<getString>` funktioniert immer (*type casting*).

- `<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";
  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()) {
      // Verarbeitung der Ergebnismenge
   String s = rset.getString(1);
   int i = rset.getInt("Population");
   System.out.println (s + " " + i + "\n");
   }
  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:

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`
`java.sql.Date`	`DATE` (Tag, Monat, Jahr)
`java.sql.Time`	`TIME` (Stunde, Minute, Sekunde)

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>
- `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 der *aktuellen Zeile* eines ResultSets

```
ResultSetMetaData rsetmetadata = rset.getMetaData();
int numCols = rsetmetadata.getColumnCount();
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.*;
class jdbcSelect {
  public static void main (String args [])
   throws SQLException {
 DriverManager.registerDriver(new
         oracle.jdbc.driver.OracleDriver());
 String url = "jdbc:oracle:thin:/*hier korrekt fortsetzen
 Connection conn =
    DriverManager.getConnection(url,"scott","tiger");
 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 returnValue = rset.getString(i);
     if (rset.wasNull()) System.out.print("null");
     else System.out.print(returnValue);
     System.out.print("  ");
  }
  System.out.println();
 }
 conn.close();
}}
```

[Filename: Java/jdbcSelect.java]

```
dbis@c42> java jdbcSelect City
```

TRANSAKTIONSSTEUERUNG

Sei `con` eine Instanz der Klasse `Connection`.
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.

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 ausgeführt werden soll.
- Abhängig von `<string>` ist nur eine der (parameterlosen!) Methoden
  - `<prepared-statement>.executeQuery()`,
  - `<prepared-statement>.executeUpdate()` oder
  - `<prepared-statement>.execute()`

  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.

```
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.*;
class jdbcCountryPop {
  public static void main (String args [])
   throws SQLException {
 DriverManager.registerDriver(new
         oracle.jdbc.driver.OracleDriver());
 String url = "jdbc:oracle:thin:/*hier korrekt fortsetzen
 Connection conn =
    DriverManager.getConnection(url,"scott","tiger");

 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.print("null");
  else System.out.print(pop);
  }
 else System.out.print("Kein zulaessiger Landescode");
 System.out.println();
 conn.close();
}}
```

[Filename: Java/jdbcCountryPop.java]

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

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

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

oder

```
<callable-statement>.executeUpdate();
```

oder

```
<callable-statement>.execute();
```

Im Beispiel:

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

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

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

Beispiel: CallableStatement

```
import java.sql.*;
class jdbcCallProc {
  public static void main (String args [])
   throws SQLException {
 DriverManager.registerDriver(new
         oracle.jdbc.driver.OracleDriver());
 String url = "jdbc:oracle:thin:/*hier korrekt fortsetzen
 Connection conn =
    DriverManager.getConnection(url,"scott","tiger");

 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 287) für den User scott/tiger verfügbar ist:

```
dbis@c42> java jdbcCallProc Joe
```

SEQUENTIELLE AUSFÜHRUNG

- SQL-Statements, die mehrere Ergebnismengen zurückliefern:
- `<statement>.execute(<string>)`, `<prepared-statement>.execute()`, `<callable-statement>.execute()`
- Häufig: `<string>` dynamisch generiert.
- `getResultSet()` bzw. `getUpdateCount()`: nächsten Rückgabewert bzw. Update-Zähler abrufen.
- `getMoreResults()` und dann wieder `getResultSet()` bzw. `getUpdateCount()`: nächstes Ergebnis abrufen.

SEQUENTIELLE AUSFÜHRUNG

- `getResultSet()`: Falls nächstes Ergebnis eine Ergebnismenge ist, wird diese zurückgegeben; falls kein nächstes Ergebnis mehr vorhanden, oder nächstes Ergebnis ein Update-Zähler ist: `null` zurückgeben.
- `getUpdateCount()`: Falls nächste Ergebnis ein Update-Zähler ist, wird dieser ($n \geq 0$) zurückgegeben; falls kein nächstes Ergebnis mehr vorhanden, oder nächstes Ergebnis eine Ergebnismenge ist, wird -1 zurückgegeben.
- `getMoreResults()`: `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

```
stmt.execute(StatementSequenceWithUnknownResults);
while (true) {
    int rowCount = stmt.getUpdateCount();
    if (rowCount > 0) { // update, n Tupel geaendert
       System.out.println("Rows changed = " + count);
       stmt.getMoreResults();
       continue;
    }
    if (rowCount == 0) { // update, aber nichts geaendert
       System.out.println("No rows changed");
       stmt.getMoreResults();
       continue;
    }
    ResultSet rs = stmt.getResultSet();
    if (rs != null) {
       .....  // verarbeite Metadaten
       while (rs.next())
          { ....}  // verarbeite Ergebnismenge
       stmt.getMoreResults();
       continue;
    }
    break;
}
```

11.4 JDBC in Java Stored Procedures

In Stored Procedures verwendet man ebenfalls JDBC-Technologie mit dem serverseitigen JDBC-Treiber von Oracle (`jdbc:default:connection:`). User/Password werden natürlich nicht angegeben, 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));
    rset.close();
    pstmt.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 -u user/passwd -r getCountryData.java
```

Definition und Ausführung des Wrappers in der DB:

```
CREATE PROCEDURE getPopulation (code IN VARCHAR2) IS
  LANGUAGE JAVA
  NAME 'getCountryData.getPop(java.lang.String)';
/
```

[Filename: Java/getCountryData.sql]

... Output aktivieren:

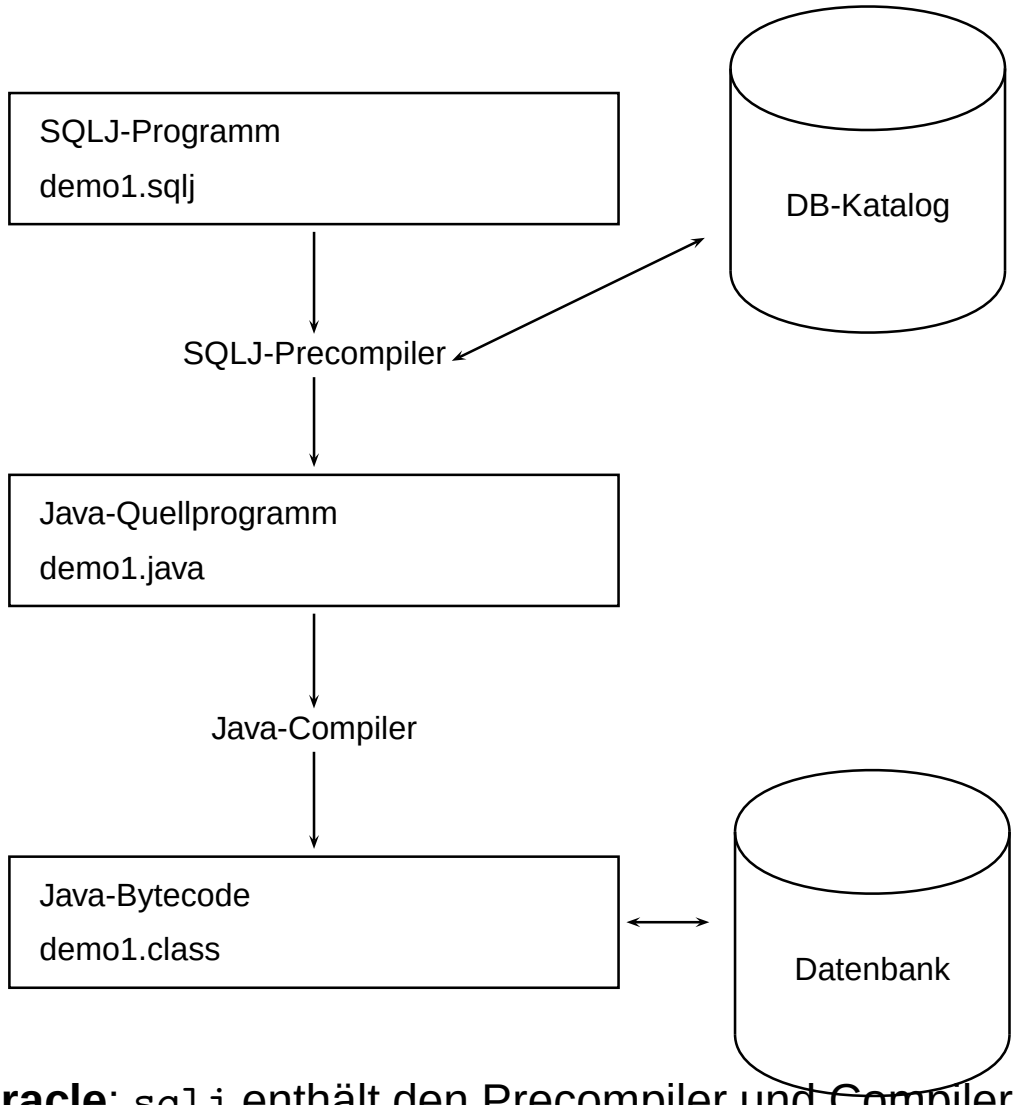
```
SET SERVEROUTPUT ON;
CALL dbms_java.set_output(2000);

EXEC getPopulation('D');
83536115
```

11.5 SQLJ

Realisierung des “Embedded SQL”-Konzeptes für Java:

- Standardisierte Java-DB Schnittstelle, verwendet JDBC
- ANSI-Standard als Teil des SQL-Standards, beteiligt waren u.a. Oracle, Sun, IBM, Microsoft (ANSI x.3.135.10-1998)
- Besteht aus drei Teilen:
  - Part 0: Embedded SQL in Java.
  - Part 1: SQL routines using Java
    (siehe Abschnitt “Java in Stored Procedures”).
  - Part 2: SQL types using Java
    (Java-Klassen als SQL Datentypen (u.a. → XMLType)).
- SQLJ Part 0: SQL-in-Java
- SQLJ Part 1 & 2: Java-in-SQL
- Eingebettete SQLJ-Aufrufe werden als pures Java übersetzt und werden auf JDBC-Aufrufe abgebildet.
- Hier: Beschränkung auf SQLJ Part 0, Part 1 als Java Stored Procedures

ENTWICKLUNG EINER SQLJ-APPLIKATION



- **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.

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";
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

- `connect.properties` ist eine Datei (bzw. Datei im .jar-Archiv), die folgendermassen aussieht:

```
#connect.properties:
sqlj.url=jdbc:oracle:thin:@xxx.xxx.xxx.xxx:1521:dbis
sqlj.user=<user>
sqlj.password=<passwd>
```

[Filename: Java/connect.properties – muss jeder selber schreiben]

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

- `<JavaClass>.class` ist eine Klasse, die im Dateisystem/jar-Archiv im selben Verzeichnis wie `connect.properties` liegt (der Name dieser Klasse dient nur dazu, `connect.properties` 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
  - Weiterschaltung mit next()
- Positionsiteratoren:
  - Spaltenzugriff über Position
  - Weiterschaltung mit FETCH . . . INTO

ITERATOREN MIT BENANNTEN SPALTEN

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

```
import java.sql.*;
import oracle.sqlj.runtime.Oracle;

class sqljIteratorExample {
 public static void main (String args []){
  // Datenbank-Verbindung aufbauen
  try {
   Oracle.connect(sqljIteratorExample.class, "connect.properties"
   // Deklaration des benannten Iterators mit Spaltennamen
   #sql iterator CountryIter(String name, int population);
   // Iteratorobjekt wird definiert
   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."); }
   // Schliessen des Iterators
   cIter.close();
  }
  catch (SQLException e) {
    System.err.println(e.getMessage()); }
 }}
```

[Filename: Java/sqljIteratorExample.sqlj]

POSITIONSITERATOREN

```
// Verbindungsaufbau, Deklaration Positionsiterator
Oracle.connect(Countries.class, "connect.properties");
#sql iterator CountryPosIterator(String, int);
// Hilfsvariablen
String name = "";
int pop = 0;

// Iteratorobjekt wird definiert
CountryPosIterator cIter;
// 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.");
}
// Schliessen des Iterators
cIter.close();
```

VERGLEICH: JDBC UND SQLJ

JDBC

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

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

SQLJ

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

```
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
- 1979: erstes Produkt
- 1992: Oracle 7
- letzte 7er: 7.3.4: erste SQLJ/JDBC-Version (SQLJ bis 8.1.5 über OTN)
- 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) `http://www.oracle.com/technology/tech/xml/xmldb/9.2.0.2.0/NewFeatures.pdf`
- 2003/2004: Oracle 10g (g = Grid)

u.a. von

`http://www.gulp.de/kb/mk/chanpos/oraclereleases.html`

(nicht auswendiglernen)

Kapitel 12
SQL und XML

12.1 XML: “Extensible Markup Language”

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

- Instantiierung von SGML (vgl. HTML)
- Semantische Tags

$\Rightarrow$ wer HTML kennt, weiß, wie XML “aussieht”.

- Baumstruktur
- Elemente (Name, Attribute und Inhalt)
- rekursiver Aufbau

$\Rightarrow$ abstrakter Datentyp mit Konstruktoren und Operationen.

- Navigation im Baum
- vgl. Pfadausdrücke in Java, OQL, SQL
  (z.B. x.coordinates.longitude)
- Pfadausdrücke in Unix:
  (z.B. /home/may/teaching/dbp/folien.tex)

$\Rightarrow$ Adressierungssprache “XPath”

XML: Beispiel

```
<country id="D" capital="cty-Germany-Stuttgart">
    <name>Germany</name>
    <total_area>356910</total_area>
    <population>83536115</population>
    <encompassed continent="europe">100</encompassed>
    <ethnicgroup name="German">95.1</ethnicgroup>
    <ethnicgroup name="Italians">0.7</ethnicgroup>
    <religion name="Roman Catholic">37</religion>
    <religion name="Protestant">45</religion>
    <language name="German">100</language>
    <border country="F">451</border>
    <border country="A">784</border>
    <border country="CZ">646</border>
    :
```

```
  <province id="prov-Germany-Baden-Wuerttemberg">
    <name>Baden Wuerttemberg</name>
    <area>35742</area>
    <population>10272069</population>
    <city is_state_cap="yes" id="cty-Germany-Stuttgart">
      <name>Stuttgart</name>
      <longitude>9.1</longitude>
      <latitude>48.7</latitude>
      <population year="95">588482</population>
    </city>
    <city id="cty-Germany-Mannheim">
      <name>Mannheim</name>
      :
    </city>
    :
  </province>
  <province id="prov-Germany-Berlin">
    <name>Berlin</name>
    <area>889</area>
    <population>3472009</population>
    <city is_country_cap="yes" is_state_cap="yes"
          id="cty-Germany-Berlin">
      <name>Berlin</name>
      <longitude>13.3</longitude>
      <latitude>52.45</latitude>
      <population year="95">3472009</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
- Basis des Semantic Web: RDF, RDF Schema, OWL ...

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 ...

SQLX UND ORACLE: LITERATUR

- Oracle TechNet (oft eine wertvolle Informationsquelle):
  `http://www.oracle.com/technology/tech/xml/index.html`
- Oracle XML DB Developer's Guide:
  `http://ap34.ifi.informatik.uni-goettingen.de/oracle-doc/appdev.102/b14259/toc.htm`
- (mit kleinen Unterschieden zum SQL/XML Standard)

12.2.1 “XML” als SQL-Objekttyp

XML/XMLType: ein vordefinierter *SQL*-Objekttyp, der XML-Daten speichert.

- Als Zeilenobjekte:

```
CREATE TABLE Mondial OF XMLType;
CREATE TABLE CountryXML OF XMLType;
```

- Als Spaltenobjekte:

```
CREATE TABLE CityXML
  (name VARCHAR2(35),
   province VARCHAR2(32),
   country VARCHAR2(3),
   population XMLType,
   coordinates XMLType);
```

[Filename: SQLX/cityxmltable.sql]

EINFACHER KONSTRUKTOR: XMLTYPE(...)

- Syntaktische Einbindung genauso wie für selbstdefinierte Objekttypen,
- XML-Inhalt als ASCII gegeben:

```
INSERT INTO tablename
   VALUES (... , XMLType('XML in ASCII-Notation') ...)
```

```
INSERT INTO cityXML
VALUES('Northeim','Niedersachsen','D',
 XMLType('<population year="2004">10000</population>'),
 XMLType('<coordinates><longitude>10</longitude>
                       <latitude>51.7</latitude>
          </coordinates>'));
```

[Filename: SQLX/cityxmltuple.sql]

IMPORT VON KOMPLETTEN XML-FILES: ALLGEMEIN

(Standardmethode, wenn es nur eine Directory gibt, aus der XML-Dokumente geladen werden sollen)

- XML-Werte können als Dateien importiert werden.
- Anlegen einer Directory, aus der die Dateien geladen werden (Admin):

```
CREATE OR REPLACE DIRECTORY XMLDIR AS '/db' ;
CREATE OR REPLACE DIRECTORY XMLDIR AS '/home/bla/...' ;
```

  Es gibt dann ein SQL Directory Object "XMLDIR".
  (Kann wie üblich mit `DROP directory XMLDIR` gelöscht werden).
- XML-File (z.B. m.xml) dorthin kopieren
  - darf keine Referenz auf eine DTD enthalten!
  - muss für alle lesbar sein: `chmod` $filename$ `644`
- Benutzung des xdb_utilities Package (muss separat installiert werden), um Files in die Datenbank zu laden:

```
INSERT INTO mondial
VALUES(xdb_utilities.getXMLfromfile('m.xml','XMLDIR'));

set long 10000 ;
SELECT * FROM mondial;
```

IMPORT VON KOMPLETTEN XML-FILES: LOKAL

- Um beliebige XML-Files zu importieren steht *lokal* im Praktikum eine Methode `system.getxml('`$http\text{-}url$`')` zur Verfügung:

```
SELECT system.getxml('
  http://www.dbis.informatik.uni-goettingen.de/
        Teaching/DBP/XML/mondial.xml') FROM dual;
```

bzw. zum Einfügen in eine Tabelle:

```
INSERT INTO mondial VALUES(
 system.getxml(
  'http://www.dbis.informatik.uni-goettingen.de' ||
  '/Teaching/DBP/XML/mondial.xml'));
```

[Filename: SQLX/insertmondial.sql]

- XML-Instanz darf keine Referenz auf eine DTD enthalten!
- z.B. aus dem eigenen Homedirectory ...
- muss für alle lesbar sein: `chmod` $filename$ `644`

Ausgabezeilenlänge anpassen

```
SET LONG 10000;
SELECT * FROM mondial;
```

12.2.2 Strukturelle XML-Konstruktoren

Der SQL/XML-Standard definiert “XML publishing functions”

- Konstruktoren des rekursiv definierten abstrakten Datentyps “XMLType”.
- Verwendung wie andere vordefinierte oder selbst definierte Funktionen (u.a. in der SELECT-Klausel).
- Erzeugen Fragmente oder Instanzen von XMLType
- Ausgabe (z.B. in SELECT) als ASCII

XMLElement

- XMLElement: Name $\times$ Element-Body $\rightarrow$ Element:
  - Element-Body: Text oder rekursiv erzeugt (Attribute und Elemente)

```
SELECT XMLElement("Leer") FROM DUAL;
```

(Ergebnis ist eigentlich nicht korrekt: `<x/>` ist ein leeres Element, während `<x></x>` ein Element mit leerem Inhalt ist!)

```
SELECT XMLElement("Country",'bla') FROM DUAL;
SELECT XMLElement(Country,'bla') FROM DUAL;
```

- Hinweis: mit “...” zur Kontrolle über Groß-/Kleinschreibung (sonst alles groß).
  (man darf hierbei auch einfache und doppelte “...” nicht anders verwenden als im ersten Beispiel).

Elemente mit Nichttrivialem Inhalt

- XMLElement: zweites Argument enthält Subknoten des Elements (Attribute, Text und Subelemente),
- XMLAttributes: Liste von Wert-Name-Paaren, aus denen Attribute erzeugt werden.

```
SELECT XMLElement("Country",
  XMLAttributes(code AS "car_code", capital AS "capital"),
  name,
  XMLElement("Population",population),
  XMLElement("Area",area))
FROM country
WHERE area > 1000000;
```

[Filename: SQLX/xmlelement.sql]

Ein Ergebnis-Element:

```
<Country car_code="R" capital="Moscow">
  Russia
  <Population>148178487</Population>
  <Area>17075200</Area>
</Country>
```

Optionale Substrukturen

- XML als abstrakter Datentyp, funktionale Konstruktoren
- semistrukturierte Daten: Flexible und optionale Substrukturen

```
SELECT XMLElement("City",
  XMLAttributes(country AS country),
  XMLElement("Name",name),
  CASE WHEN longitude IS NULL THEN NULL
       ELSE XMLElement("Longitude",longitude) END,
  CASE WHEN latitude IS NULL THEN NULL
       ELSE XMLElement("Latitude",latitude) END)
FROM city
WHERE longitude IS NOT null;
```

[Filename: SQLX/xmlelement2.sql]

- Hinweis: `CASE WHEN` $cond$ `THEN` a `ELSE` b `END` ist ein *funktionales* Konstrukt.

Einfache Elementinhalte

- XMLForest: erzeugt eine Liste einfacher XML-Elemente aus gegebenen Namen und Werten:

```
SELECT XMLElement("Country",
          XMLForest(name AS Name,
                    code AS car_code,
                    population AS "Population",
                    area AS "Area"))
FROM country
WHERE area > 1000000;
```

[Filename: SQLX/xmlforest.sql]

```
<Country>
  <NAME>Brazil</NAME>
  <CAR_CODE>BR</CAR_CODE>
  <Population>162661214</Population>
  <Area>8511965</Area>
</Country>
```

$\Rightarrow$ kanonische Abbildung von Tupeln auf einfache XML-Elemente.

Subqueries

Textinhalte können auch durch (korrelierte) Subqueries bestimmt werden:

```
SELECT XMLElement("Country",
         XMLAttributes(code AS car_code),
         XMLElement("Name",name),
         XMLElement("NoOfCities",
                    (SELECT count(*)
                     FROM City
                     WHERE country=country.code)))
FROM country
WHERE area > 1000000;

SELECT XMLElement("Country",
         XMLAttributes(code AS car_code),
         XMLElement("Name",name),
         (SELECT XMLElement("NoOfCities",count(*))
                       FROM City
                       WHERE country=country.code))
FROM country
WHERE area > 1000000;
```

[Filename: SQLX/xmlsubquery.sql]

Gruppierung: XMLAgg

- XMLAgg: Bildung einer Collection aus den Zeilen der Gruppierung nach GROUP BY:
  In XML kann man auch die *Liste* der Items innerhalb der Gruppe verwenden:

```
SELECT XMLElement("Country",
         XMLAttributes(country AS car_code),
         XMLElement("NoOfCities", count(*)),
         XMLAgg(XMLElement("city",name)
                              ORDER by population))
FROM city
GROUP BY country;
```

[Filename: SQLX/xmlagg.sql]

Element des Ergebnisses:

```
<Country CAR_CODE="D">
  <NoOfCities>85</NoOfCities>
  <city>Erlangen</city>
  <city>Kaiserslautern</city>
  :
  <city>Berlin</city>
</Country>
```

Gruppierung: XMLAgg

- ähnlich über eine (korrelierte) (Sub)query:

```
SELECT XMLElement("Country",
       XMLAttributes(name AS name),
        (SELECT XMLAgg(XMLElement("city",name))
         FROM City
         WHERE country=code))
FROM country;
```

... wobei jetzt `XMLElement(''NoOfCities'', count(*))` (ebenfalls aus der Subquery zu berechnen) fehlt.

XMLConcat

- XMLConcat: Aneinanderhängen von mehreren “Spalten” einer Anfrage zu einem XML-Fragment:

```
SELECT XMLElement("Country",
  XMLAttributes(code AS code),
  XMLELEMENT(name, name),
  (SELECT XMLConcat(
     XMLElement("NoOfCities", count(*)),
     XMLAgg(XMLElement("city",name)))
   FROM City
   WHERE country=code))
FROM country;
```

[Filename: SQLX/xmlconcat.sql]

Auf diese Weise erzeugtes XML wird verwendet, um Instanzen von XMLType in Tabellen zu erzeugen:

XML-Zeilenobjekte

```
CREATE TABLE CountryXML OF XMLType;

INSERT INTO CountryXML
(SELECT XMLElement("Country",
  XMLAttributes(code AS "Code",
                population AS "Population"),
  XMLElement("Name",name),
  (SELECT XMLElement("Capital",
            XMLForest(name AS "Name",
                      population AS "Population"))
   FROM city
   WHERE country=country.code
     AND city.name=capital))
FROM country);
```

[Filename: SQLX/fillcountry.sql]

- Ergebnis des SELECT sind Objekte vom Typ XMLType.

XML-Spaltenobjekte

```
CREATE TABLE CityXML
  ( name VARCHAR2(35),
    province VARCHAR2(32),
    country VARCHAR2(3),
    population XMLType,
    coordinates XMLType);

INSERT INTO CityXML
(SELECT name, province, country,
  XMLElement("Population",
             XMLAttributes(95 as year),
             population),
  CASE WHEN longitude IS NULL THEN NULL
       ELSE XMLElement("Coordinates",
              XMLElement("Longitude", longitude),
              XMLElement("Latitude", latitude))
   END
 FROM city);
```

[Filename: SQLX/fillcity.sql]

... so weit zum Erzeugen und Abspeichern von XML-Daten.

12.2.3 Anfragen an XML-Daten innerhalb SQL

- XMLType als Abstrakter Datentyp: Selektoren, Modifikatoren.
- Diese bieten Schnittstellen für Standard-XML-Sprachen.
- Signatur:

  extract: XMLType $\times$ XPath_Expression $\rightarrow$ XMLType/text/number

  extractValue: XMLType $\times$ XPath_Expression $\rightarrow$ text/number

  existsNode: XMLType $\times$ XPath_Expression $\rightarrow$ Boolean

- als “freie” Methoden (XMLType-Objekt mit `value` selektieren):

```
SELECT extract(value(m), '//city[name="Berlin"]')
FROM mondial m;
```

- oder als “member methods” wie bei benutzerdefinierten Objekttypen:

```
SELECT m.extract('//city[name="Berlin"]')
FROM mondial m;
```

SELECT: "EXTRACT"-FUNKTION

extract(XMLType_instance, XPath_string)
XMLType_instance.extract(XPath_string)

- Erstes Argument: SQL – selektiert ein (SQL-)Attribut des gegenwärtigen Tupels (muss ein Objekt des Typs XMLType ergeben),
- Zweites Argument: wendet einen XPath-Ausdruck darauf an,
- Ergebnis: vom Typ XMLType oder anderer SQL-Typ (mehrwertige Ergebnisse werden aneinandergehängt).

XML-ZEILENOBJEKTE

Zeilenwert ist vom Typ XMLType:

```
SELECT extract(value(c), '/Country/@CODE'),
       extract(value(c), '/Country/Capital/Name')
FROM CountryXML c;

SELECT c.extract('/Country/@CODE'),
       c.extract('/Country/Capital/Name')
FROM CountryXML c;
```

EINSCHUB: KURZÜBERSICHT ÜBER XPATH

- Navigation wie in Unix: /*step*/*step*/*step*
  /mondial/country/name
- Groß-Kleinschreibung beachten!
- Ergebnis: Menge/Liste von XML-Knoten (können nicht nur Werte, sondern ganze Teilbäume sein):
  /mondial/country
- Schritte überspringen:
  /mondial//city/name
  //city/name
  (ergibt /mondial/country/city und
  /mondial/country/province/city)
- Attribute: .../@*attributname*:
  /mondial/country/@area
- Zugriff auf Textinhalt:
  /mondial/country/name/text()
- Tests während der Navigation:
  /mondial/country[@code='D']/@area
  /mondial/country[name/text()='Germany']/@area
- Bei Vergleichen wird automatisch der Textinhalt verwendet:
  /mondial/country[name='Germany']/@area

(Weitere Details und Systematik siehe XML-Vorlesung)

SELECT: "EXTRACT"-FUNKTION (FORTS.)

XML-SPALTENOBJEKTE

Die Werte von CityXML.population sind XMLType-Zeilenobjekte:

```
SELECT extract(population,'/') FROM CityXML;
SELECT c.population.extract('/') FROM CityXML c;

SELECT name,
   extractValue(population,'/Population/@YEAR'),
   extractValue(population,'/Population')
FROM CityXML;

SELECT name,
 c.population.extract('/Population/@YEAR').getNumberVal(),
 c.population.extract('/Population/text()').getNumberVal()
FROM CityXML c
ORDER BY 3;
```

- exakte Groß-/Kleinschreibung im XPath-Ausdruck,
- extractValue momentan nicht als member method implementiert (Fehler)
- Benutzung von getNumberVal() und getStringVal()-Funktionen zum Casting:
  XML kennt soweit keinen Unterschied zwischen Strings und numerischen Werten.

SUBQUERIES AN XMLTYPE IN DER WHERE-KLAUSEL

... zum Heraussuchen und Vergleichen ebenfalls mit extract():

```
SELECT name
FROM CityXML c
WHERE c.population.extract('/Population/text()')
        .getNumberVal() > 1000000;

SELECT c.extract('/Country/Name/text()')
FROM CountryXML c
WHERE c.extract('/Country/@Population')
        .getNumberVal() > 1000000;
```

- Hierbei findet der Vergleich auf der SQL-Ebene statt (Vorteil: man kann Joins bilden).
- Hinweis: Wenn der XPath-Ausdruck mehrere Ergebnisse liefert, werden diese (bereits bei der Auswertung der extract()-Funktion) aneinandergehängt ...
- ... dann muss man es anders machen.

"EXISTSNODE"-FUNKTION

existsNode(XMLType_instance, XPath_string)

- Erstes Argument: SQL – selektiert ein (SQL-)Attribut des gegenwärtigen Tupels (muss ein Objekt des Typs XMLType ergeben),
- Zweites Argument: testet ob der angegebene XPath-Ausdruck für das Objekt ein nichtleeres Ergebnis hat
- Vergleich findet für jeden betroffenen Knoten einzeln auf XML-Ebene statt.
  Der Vergleichwert muss im XPath-String angegeben werden, man kann also damit *keine* Joins bilden.
- Ergebnis: 1 falls es einen Knoten gibt, 0 sonst.

```
SELECT name, extractValue(Population,'/')
FROM CityXML
WHERE existsNode(population,
                 '/Population[text()>1000000]') = 1;

SELECT name, extractValue(Population,'/')
FROM CityXML c
WHERE c.population.
      existsNode('/Population[text()>1000000]') = 1;
```

12.2.4 Ändern von XML-Daten

- das XMLType-Objekt wird komplett *ersetzt*,
- updateXML(...) als (Transformations-)Funktion:

  updateXML(XMLType_instance, XPath_string, new_value)
- Erstes Argument: SQL – selektiert ein (SQL-)Attribut des gegenwärtigen Tupels (muss ein XMLType-Objekt sein),
- $2n$-tes Argument: selektiert den/die zu ändernden Knoten durch einen XPath-Ausdruck,
- $2n+1$-tes Argument: neuer Wert,
- Ergebnis: geänderte Instanz vom Type XMLType.
- Der Ausdruck "SELECT updateXML(...) FROM ..." ändert nicht die Datenbank, sondern gibt nur den Wert aus, der aus dem Update resultieren würde.

```
SELECT updateXML(c.population,
                 'Population/text()','1000000',
                 'Population/@YEAR','2004')
FROM CityXML c WHERE name='Gottingen';

SELECT updateXML(value(c),
                 '/Country/Name/text()','Fidschi')
FROM CountryXML c
WHERE extractValue(value(c),'Country/Name')='Fiji';
```

[Filename: SQLX/updatexml.sql]

Ändern von XML-Daten

Diese Funktion wird dann im SET-Statement verwendet:

```
UPDATE CityXML c
SET c.population   -- an XMLType element
    = updateXML(c.population,
                'Population/text()','1000000',
                'Population/@YEAR','2004')
WHERE name='Gottingen';

UPDATE CountryXML c
SET value(c) = updateXML(value(c),
                '/Country/Name/text()','Fidschi')
WHERE extractValue(value(c),'Country/Name')='Fiji';

UPDATE CountryXML c
SET value(c) = updateXML(value(c),
                '/Country/Name/text()','Fidschi')
WHERE existsNode(value(c),'/Country[Name="Fiji"]') = 1;
```

12.2.5 XML-Spezifische Funktionalität

Member Methods von XMLType

- Anwenden von XSLT-Stylesheets auf XMLType

Java-Funktionalität und PL/SQL Packages

- Implementierungen vieler XML-Sprachen mitgeliefert,
- dbms_xmldom: implementiert DOM (API um direkt mit der Baumstruktur zu arbeiten):
  PL/SQL: dbms_xmldom.*dosomething*(*object,args*)
- dbms_xmlparser: parst XML-Dokumente und DTDs (gegeben als CLOB oder URL) uns speichert das Ergebnis: Zugriff auf die DOM-Instanz und die DTD innnerhalb des Parsers durch “getdocument” bzw. “getdoctype”
- dbms_xslprocessor: processxsl(*verschiedene Argumente*); clob2file/file2clob liest/schreibt;
  selectnodes/selectsinglenode/valueof: XPath-Anfragen

... Details: Oracle Dokumentation, google ...

XSLT in Oracle: "Transform" Member Method

Member Method von XMLType:
XML-instance.transform(*Stylesheet-as-XMLValue*)
als SQL-Funktion anwendbar: SELECT
XMLTransform(*XML-instance*,*Stylesheet-als-XMLType*)

```
CREATE TABLE stylesheets
 (name VARCHAR2(100),
  stylesheet XMLTYPE);

INSERT INTO stylesheets VALUES('mondial-simple.xsl',
 system.getxml(
  'http://www.dbis.informatik.uni-goettingen.de' ||
   '/Teaching/DBP/XML/mondial-simple.xsl'));

SELECT value(m).transform(s.stylesheet)
FROM mondial m, stylesheets s
WHERE s.name = 'mondial-simple.xsl';

SELECT XMLTransform(value(m),s.stylesheet)
FROM mondial m, stylesheets s
WHERE s.name = 'mondial-simple.xsl';
```

[Filename: SQLX/applystylesheet.sql]

```
CREATE OR REPLACE FUNCTION xslexample RETURN CLOB IS
  xmldoc      CLOB;   xsldoc  CLOB;  html  CLOB;
  myParser    dbms_xmlparser.Parser;
  indomdoc    dbms_xmldom.domdocument;
  xsltdomdoc  dbms_xmldom.domdocument;
  xsl         dbms_xslprocessor.stylesheet;
  outdomdocf  dbms_xmldom.domdocumentfragment;
  outnode     dbms_xmldom.domnode;
  proc        dbms_xslprocessor.processor;
BEGIN
  -- Get the XML document as CLOB
  SELECT value(m).getClobVal() INTO xmldoc FROM mondial m;
  -- Get the XSL Stylesheet as CLOB
  SELECT s.stylesheet.getClobVal() INTO xsldoc
  FROM stylesheets s WHERE name='mondial-simple.xsl';

  -- Get the new xml parser instance
  myParser := dbms_xmlparser.newParser;
  -- Parse the XML document and get its DOM
  dbms_xmlparser.parseClob(myParser, xmldoc);
  indomdoc := dbms_xmlparser.getDocument(myParser);

  -- Parse the XSL document and get its DOM
  dbms_xmlparser.parseClob(myParser, xsldoc);
  xsltdomdoc := dbms_xmlparser.getDocument(myParser);

  xsl  := dbms_xslprocessor.newstylesheet(xsltdomdoc, '');
  -- Get the new xsl processor instance
  proc := dbms_xslprocessor.newProcessor;

  -- Apply stylesheet to DOM document
  outdomdocf := dbms_xslprocessor.processxsl(proc, xsl, indomdoc);
  outnode    := dbms_xmldom.makenode(outdomdocf);

  -- Write the transformed output to the CLOB
  dbms_xmldom.writetoCLOB(outnode, html);
  return(html);  -- Return the transformed output
END;
/
```

[Filename: SQLX/xslexample.sql]