

Objektrelationale Erweiterungen von SQL2003

6.1 Objektrelationales SQL	299
6.2 Objektrelationale Anwendungsprogrammierung: JDBC und SQLJ	322
Zusammenfassung	345
Weiterführende Literatur	345
Übungsaufgaben	346

» Dieses Kapitel ist ein Drahtseilakt, denn einerseits gibt es in SQL2003 eine Menge neuer objektrelationaler Konstrukte, die bei weitem nicht alle in einem Kapitel beschrieben werden können. Andererseits sind diese noch selten bei den Datenbankherstellern nach dem Standard implementiert. Die einzelnen Hersteller bieten sehr verschiedene Sprachbestandteile an. Objektrelationales SQL im Oracle-Dialekt kann ein ganzes Buch¹ füllen, in MySQL sind diese Erweiterungen noch sehr rudimentär bis gar nicht vorhanden (vgl. Abschnitt 6.1.9 und Abschnitt 6.1.10). Der Umfang der objektrelationalen Erweiterungen insgesamt ist schon viel zu groß, um ihn umfassend in einem einführenden Lehrbuch über Datenbanksysteme zu behandeln. Daher wurde versucht, hier eine funktionsfähige, relevante Auswahl zu treffen, die keinen Anspruch auf Vollständigkeit erhebt. So fehlen objektrelationale Sichten, Methoden von benutzerdefinierten Typen und Tabellenhierarchien, um nur einige Punkte zu nennen. Auch die Syntaxdefinitionen sind nicht vollständig, aber funktionsfähig. Oft wurden stattdessen nur Beispiele aufgeführt, die einen Einblick in die Konstrukte geben sollen.

Im Bereich objektrelationaler Datenbankprogrammierung sieht es etwas einfacher aus. Dort sind der JDBC-Standard sowie der SQLJ-Standard allseits bekannt und akzeptiert und sie werden hier in den Grundzügen vorgestellt. Beide Standards sind in SQL2003 eingeflossen und zeigen, wie sich eine objektorientierte Programmiersprache mit relationalen Datenbanken verbinden lässt. Im Bereich JDBC wird der Standardaufbau eines Programms mit Datenbankanbindung vorgestellt. Ebenso werden die Themen Transaktionsverwaltung, Fehlerbehandlung und Metadaten angesprochen. Auf dem Gebiet SQLJ werden SQLJ-Klauseln, Host-Variablen, Iteratoren und Kontexte eingeführt.

Falls kein objektrelationales Datenbanksystem zur Verfügung steht, können zu einem objektrelationalen Datenmodell Standardabbildungen definiert werden, die den Entwurf auf ein relationales Datenbankschema definieren. Diese Abbildungen heißen objektrelationale Abbildungen. Das Vorgehen ist in der Praxis weit verbreitet, da sich objektrelationale Modelle gut zur Beschreibung komplexer Probleme eignen und man den Umstieg auf ein objektrelationales System noch scheut. «

¹ z.B. [Hohenstein 2002]

Ziele

Nach dem Lesen dieses Kapitels und dem Lösen der Übungsaufgaben werden Sie in der Lage sein,

- das objektrelationale Datenmodell mit seinen neuen Basisdatentypen, Typkonstruktoren und Regeln zu verstehen und anzuwenden,
- benutzerdefinierte Datentypen unter SQL2003 zu definieren,
- Tupeltabellen und typisierte Tabellen zu benutzen,
- Vererbungshierarchien von benutzerdefinierten Typen aufzubauen,
- Anfragen an objektrelationale Tabellen zu stellen,
- den objektrelationalen Oracle-Dialekt anzuwenden,
- eine objektrelationale Abbildung von einem objektrelationalen Modell auf ein relationales Datenmodell durchzuführen,
- Datenbankverbindungen mittels JDBC aufzubauen und einfache JDBC-Programme zu realisieren, sowie zu verstehen, welche Unterschiede es zwischen JDBC und SQLJ gibt und wie SQLJ-Programme benutzt werden.

6.1 Objektrelationales SQL

6.1.1 Anwendungsfelder objektrelationaler Datenbanken

Objektrelationale Datenbanksysteme sind zurzeit (2007) Stand der Technik, Stonebraker bezeichnete sie schon in 1999 im Titel seines Buchs „Objektrelationale Datenbanken“ als „die nächste große Welle“. Er nahm dort² auch eine schon grobe Einteilung des Einsatzes von objektrelationalen Datenbanksystemen vor:

Einfache Daten und viele Anfragen erfordern Relationale Datenbanksysteme	Komplexe Daten und viele Anfragen erfordern Objektrelationale Datenbanksysteme
Einfache Daten und keine Anfragen erfordern Dateisysteme	Komplexe Daten und keine Anfragen erfordern Objektorientierte Datenbanksysteme

Seit 1997 sind die ersten objektrelationalen Datenbanksysteme auf dem Markt, mit SQL2003 wurden sie 1999 und dann 2003 erstmalig standardisiert. Thema dieses Kapitels sind die objektrelationalen Datenbanksysteme wegen ihrer großen, anwachsenden Verbreitung. Die Einsatzgebiete von objektrelationalen Datenbanksystemen weisen einige Besonderheiten auf:

- Es handelt sich um besonders komplexe Objekte:
 - z.B. technische Objekte, die aus CAD-Zeichnungen entstehen,
 - geografische Objekte,

² [Stonebraker et al. 1999, S. 2]

- Datenreihen und Ergebnisse von Zeitreihenanalysen oder
- Ergebnisse medizinischer Untersuchungen.
- Es handelt sich um besonders große Objekte:
 - z.B. Luftbildaufnahmen oder Bildersammlungen,
 - Videoaufzeichnungen,
 - aber auch lange Texte.
- Die Objekte haben besondere Eigenschaften oder Operationen:
 - bei geografischen Objekten: schneidet, liegt in, Entfernung messen
 - bei Bildern: anzeigen, zuschneiden, Farben verändern, Ähnlichkeiten suchen
 - bei Videoaufnahmen: abspielen, zuschneiden.

Verglichen mit den relationalen Datenbanksystemen handelt es sich um eine Anwendung der bewährten Datenbanktechnik auf Non-Standard-Daten, wobei die Vorteile von traditionellen Datenbanken, wie Sicherheit und Unterstützung des Multi-User-Betriebs, erhalten bleiben. Türker und Saake beschreiben diese Entwicklung³ als evolutionäre Erweiterung relationaler DBMS durch die Integration objektorientierter Konzepte. Ziel ist es dabei, eine Aufwärtskompatibilität basierend auf SQL und eine Vereinigung der Vorteile relationaler und objektorientierter DBMS zu erreichen.

Der Übergang ging evolutionär vonstatten, bei Türker⁴ wird beschrieben, welche Zwischenschritte und Modelle dabei entstanden sind. Wir betrachten hier nur das Endergebnis: das objektrelationale Typsystem als Grundlage eines objektrelationalen Datenmodells. Ein Datenmodell⁵ besteht dabei aus Basisdatentypen, Typkonstruktoren und Typkonstruktionsregeln. Die Basisdatentypen sind im Wesentlichen unverändert gegenüber den relationalen Typen, einzelne Neuerungen werden im nächsten Kapitel vorgestellt.

6.1.2 Basisdatentypen

In SQL1999 sind die Basisdatentypen BOOLEAN, BLOB und CLOB hinzugekommen, in SQL2003 zusätzlich noch XML und BIGINT. In Zusammenhang mit den objektrelationalen Erweiterungen behandeln wir in diesem Kapitel die Multimediatypen BLOB und CLOB sowie XML.

Mit der Anweisung

```
CREATE TABLE Angestellte1 (Angestellten_NR NUMERIC,
                            Nachname      VARCHAR(30),
                            BILD          BLOB(2M),
                            Lebenslauf    CLOB(2M));
```

wird ein Angestellter mit einem BLOB (Binary Large Object) und einem CLOB (Character Large Object) angelegt. Beide Datentypen müssen eine Größenangabe enthalten. BLOB- und CLOB-Spalten können weder in Primärschlüsseln, Zweitenschlüsseln noch in der GROUP-BY-Klausel und der ORDER-BY-Klausel einer SELECT-Anweisung verwendet werden.

³ [Türker et al. 2006]

⁴ [Türker et al. 2006, S. 7 ff. und S. 43 ff.]

⁵ vgl. Kapitel 4

Der Lebenslauf kann auch als XML-Dokument angelegt werden:

```
CREATE TABLE Angestellte2 (Angestellten_NR NUMERIC,
                            Nachname      VARCHAR(30),
                            BILD          BLOB(2M),
                            Lebenslauf    XML);
```

Im SQL-Standard sind noch eine Anzahl von XML-Funktionen enthalten, die aber weder unter Oracle noch unter MySQL vollständig implementiert sind, z.B. XMLGEN, XMLELEMENT, XMLFOREST XMLCONCAT und XMLAGG⁶. Auf die Besonderheiten von BLOB und CLOB sowie XML unter Oracle und MySQL wird in den Abschnitten 6.1.9 und 6.1.10 eingegangen.

6.1.3 Objektrelationale Typkonstruktoren und Regeln

In Kapitel 4 und Kapitel 5 führten wir schon Datenmodelle allgemein ein, insbesondere das relationale Datenmodell und das SQL-Datenmodell. Während die Basisdatentypen im Wesentlichen gleich bleiben, ist dies bei den Typkonstruktoren nicht der Fall. Neu hinzugekommen sind beim objektrelationalen Modell die Typkonstruktoren LIST, ARRAY, OBJECT und REF sowie verschiedene Typkonstruktionsregeln.⁷

- OBJECT beschreibt ein Objekt.
- REF beschreibt eine Referenz auf ein Objekt.
- LIST beschreibt eine geordnete Menge von gleichartigen Elementen, deren Anzahl nicht festliegt.
- ARRAY beschreibt eine geordnete Menge von gleichartigen Elementen, deren Anzahl festliegt.

Wesentlich sind hier nicht nur die Einführung der neuen Typkonstruktoren, sondern auch die unterschiedlichen Typkonstruktionsregeln. So können die unterschiedlichen Typkonstruktoren beinahe beliebig aufeinander angewendet werden, was eine Vielzahl von Erweiterungen zur Folge hat, von denen wir ihm Rahmen dieses Lehrbuchs nur einige wenige vorstellen können.

Nach Türker⁸ sieht das objektrelationale Datenmodell, welches SQL2003 zugrunde liegt, dann wie in ► Abbildung 6.1 aus.

Der Typkonstruktor MULTISET ist als Typkonstruktor neben REF und ARRAY erst in SQL2003 enthalten. ARRAY und MULTISET nehmen beide eine Menge von Werten auf, was eine gewollte Verletzung der ersten Normalform bedeutet. MULTISET taucht im Diagramm zweimal auf, einmal links als Einstiegspunkt in die Datenbank (links, gestrichelte Linie), was dem SQL-Datenmodell entspricht. Und einmal im unteren Teil als zusätzlicher Typkonstruktor, der in einer ROW oder einem OBJECT verwendet werden kann.

⁶ Nähere Einzelheiten sind in [Türker 2003, S. 438] oder <http://www.wiscorp.com/SQL2003Features.pdf>, 10.12.2006 enthalten.

⁷ Datenmodelle mit Typkonstruktoren und Typkonstruktionsregeln wurden bereits im Abschnitt 4.1 beschrieben.

⁸ vgl. [Türker 2003, S. 102]

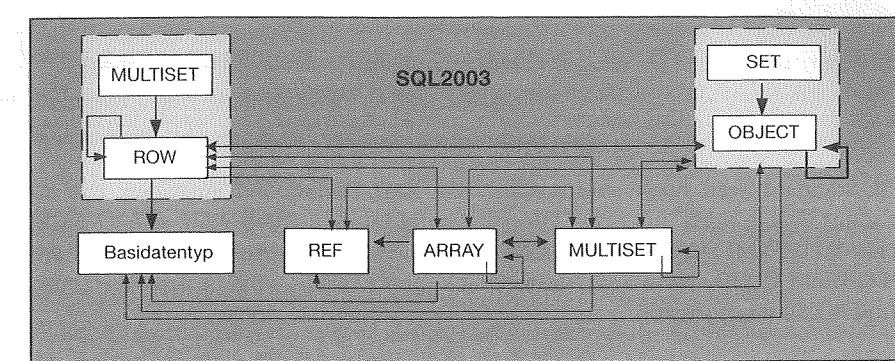


Abbildung 6.1: Typkonstruktoren SQL2003

6.1.4 Tupeltabellen

Je nach Einstiegspunkt in die Grafik ▶ 6.1 handelt es sich um Tupeltabellen, wenn MULTISSET (ROW(...)), also der linke Teil der oben angegebenen Abbildung, gebildet wird, oder um typisierte Tabellen, die dem rechten Teil der Grafik (SET(OBJECT(...))) entsprechen.

In der CREATE TABLE-Anweisung sind bei den Tupeltabellen neben den Standardbasisdatentypen auch die unbenannten Typkonstruktoren ROW, REF, ARRAY und MULTISSET zugelassen.

Die folgende Abbildung⁹ zeigt zusammenfassend, welche Spaltentypen in Tupeltabellen auftreten können:

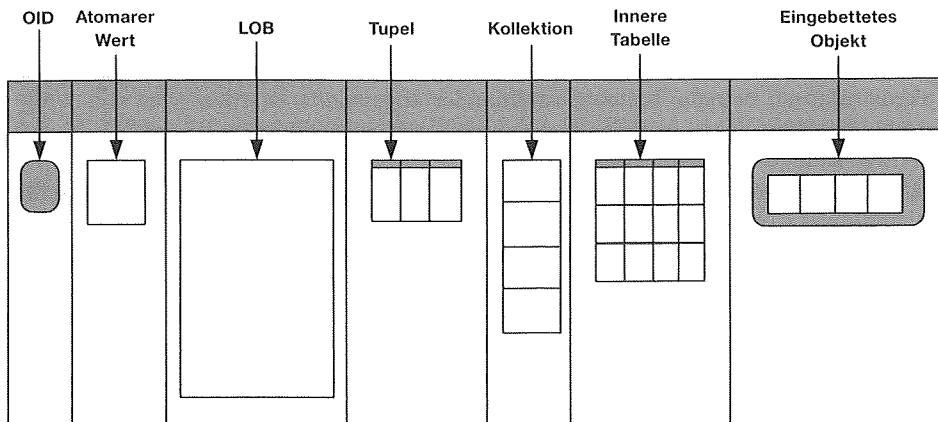


Abbildung 6.2: Objektrelationale Tupeltabellen

Durch die Erweiterung um die Basisdatentypen und Typkonstruktoren kann eine objektrelationale Tupeltable als Spaltendefinition neben den atomaren Werten, die schon in SQL2 üblich waren, noch enthalten:

- LOBS, also Spalten für Large Objects (BLOB, CLOB, NCLOB)
- Tupel, die aus anderen Datentypen zusammengesetzt sind

⁹ Die Grafik ist aus [Türker et al. 2006, S. 122] entnommen.

- Kollektionen, d.h. Mengen von Werten (SET, MULTISET)
- Innere Tabellen, d.h. tabellenwertige Spalten
- Eine OID als Objektidentifizierer
- Eingebettete Objekte

Eine Vertiefung dieser Thematik ist hier leider nicht möglich¹⁰. Das Thema kann nur in groben Zügen und anhand von Beispielen behandelt werden.

Beispiel

```
CREATE TABLE Angestellte3
  (Angestellten_Nr      NUMERIC,
   Adresse (ROW(STRASSE  VARCHAR(20),
              NR        NUMERIC,
              PLZ       NUMERIC(5),
              ORT       VARCHAR(20))),
```

Damit ist implizit ein zusammengesetzter Datentyp definiert, der unbenannt bleibt und die Adresse beschreibt.

Der REF-Typkonstruktor verweist auf eine typisierte Tabelle, die im Abschnitt 6.1.6 behandelt wird. Referenzen können nur auf typisierte Tabellen, die eine OID besitzen, verweisen.

Beispiel

```
CREATE TABLE Angestellte4
  (Angestellten_NR      NUMERIC,
   Adresse (ROW(Strasse  VARCHAR(20),
              NR        NUMERIC,
              PLZ       NUMERIC(5),
              ORT       VARCHAR(20))),
   Abteilung REF(AbteilungsTyp) SCOPE Abteilungen);
```

Die Spalte „Abteilung“ enthält eine Referenz auf die typisierte Tabelle „Abteilung“, die dem Typ „Abteilungtyp“ entspricht.

Beispiel

```
CREATE TABLE Angestellte5
  (Angestellten_NR      NUMERIC,
   Adresse (ROW(Strasse  VARCHAR(20),
              NR        NUMERIC,
              PLZ       NUMERIC(5),
              ORT       VARCHAR(20))),
   Abteilung REF(AbteilungsTyp) SCOPE Abteilungen,
   Telefonnummer        VARCHAR(20) ARRAY[3]);
```

Ein Angestellter kann bis zu drei Telefonnummern als Kollektionstyp haben, die auch eine Reihenfolge haben, die durch den Index gegeben ist.

¹⁰ Wer an einer ausführlichen Darstellung dieser Thematik interessiert ist, wird auf [Türker 2003] und [Türker et al. 2006] verwiesen.

Beispiel

```
CREATE TABLE Angestellte6
  (Angstellten_NR      NUMERIC,
   Adresse (ROW(Strasse  VARCHAR(20),
                NR       NUMERIC,
                PLZ      NUMERIC(5),
                ORT      VARCHAR(20))),
   Abteilung REF(AbteilungsTyp) SCOPE Abteilungen,
   Telefonnummer        VARCHAR(20) ARRAY[3],
   Hobbies              VARCHAR(20) MULTISET);
```

MULTISSET ist hier eine ungeordnete Menge von beliebigen Elementen, die auch Duplikate erlaubt und deren Anzahl nicht festliegt. Es sind noch eine Vielzahl von Konstruktionen möglich, da die Typkonstruktionsregeln beliebig aufeinander angewendet werden können. So kann insbesondere auch eine geschachtelte Tabelle, also eine Tabelle mit tabellenwertigen Attributen, entstehen, wenn man zweimal den Typkonstruktor MULTISSET anwendet:

```
MULTISSET(ROW( MULTISSET(ROW(Basisdatentyp))))
```

Unter Oracle sind diese unbenannten Typkonstruktoren bis auf den REF-Konstruktor nicht vorgesehen, stattdessen sind benannte Typkonstruktoren (vgl. Abschnitt 6.1.9) möglich. MySQL besitzt noch so gut wie keine objektrelationalen Erweiterungen.

6.1.5 Strukturierte Typen und Vererbung

Eine wesentliche Neuerung in SQL1999 ist die Möglichkeit, neue Typen als benutzerdefinierte Datentypen anzulegen, mit denen auch eine Vererbungshierarchie aufgebaut werden kann.

```
<CREATE TYPE Anweisung> ::=  
  CREATE TYPE Typname [ UNDER Supertypname ] AS  
    ( <Attributdefinition> [, <Attributdefinition> ]... )  
    [ NOT ] INSTANTIABLE  
    [ NOT ] FINAL  
    [ ,<OID-Typspezifikation> ]  
    [ <Methodenspezifikation> [, <Methodenspezifikation> ]... ];
```

Die Attributdefinition enthält einen Attributnamen und einen Datentyp, der auch selbst definiert, d.h. über eine CREATE TYPE-Anweisung definiert sein kann. Mit INSTANTIABLE wird festgelegt, dass man von diesem Typ eine Instanz (heißt hier typisierte Tabelle) bilden kann. Mit UNDER <Supertypname> erbt der neu definierte Typ Attribute und Methoden des Supertyps. Mit NOT FINAL wird festgelegt, dass vom Typ Subtypen gebildet werden können. Typen, die keinen Supertyp besitzen, heißen Wurzeltypen. Die <OID-Typspezifikation> hat folgende Varianten:

- REF IS SYSTEM GENERATED: Die OID wird vom System generiert.
- REF FROM (<Attributliste>): Die OID wird aus der Attributliste abgeleitet.
- REF USING (<Datentyp>): Die OID basiert auf einem Basisdatentypen (außer BLOB oder CLOB).

Die Methodendeklaration benutzt PSM¹¹ und kann sowohl INSTANCE-Methoden, CONSTRUCTOR-Methoden als auch STATIC-Methoden enthalten. Die entsprechende Oracle-Syntax ist im Abschnitt 6.1.1 enthalten und von PSM verschieden. Auf die Darstellung der PSM-Funktionalität¹² zur Definition von Methoden wurde verzichtet, da es noch kein Datenbanksystem gibt, welches sie vollständig umsetzt.

Beispiel

```
CREATE TYPE Adresse_t AS
  (Strasse  VARCHAR(30),
   NR       INTEGER,
   PLZ      INTEGER,
   Ort      VARCHAR(20),
   LAND     VARCHAR(50)) NOT FINAL;

CREATE TYPE Person_t AS
  (P_Nr      INTEGER,
   Nachname  VARCHAR(50),
   Vorname   VARCHAR(50),
   Geschlecht CHAR(1),
   Adresse   Adresse_t,
   Zeitstempel DATE) NOT FINAL;

CREATE TYPE Angestellter_t UNDER Person_t
  (Aufgabenbeschreibung CLOB,
   Beruf      VARCHAR(10),
   Eintrittsdatum DATE,
   Gehalt     NUMERIC(5,2),
   Abzuege    NUMERIC(5,2)) NOT FINAL
   REF IS SYSTEM GENERATED;

CREATE TYPE Lieferanten_t UNDER Person_t
  (Telefonnummer  VARCHAR(30),
   Beschreibung   VARCHAR(30));
```

Der Typ Person_t verwendet den benutzerdefinierten Datentyp Adresse_t als Datentyp. Der Typ Angestellter_t erbt als Subtyp vom Supertyp Person_t alle Attribute und (die nicht vorhandenen) Methoden. Lieferanten_t ist ein zweiter Subtyp vom Supertyp Person_t.

Zu jedem benutzerdefinierten Datentyp gibt es eine Konstruktormethode, die den Namen des Typs hat und als Parameter die Attributnamen des Typs verwendet. Diese Methode wird verwendet in INSERT-, UPDATE- und DELETE-Anweisungen.

Beispiel

für eine Konstruktormethode:

```
Adresse_t('Waldweg', '10', 51643, 'Gummersbach', 'Deutschland')
```

¹¹ vgl. [DATE et al. 1998], PSM steht für Persistent Stored Modul und wurde mit den SQL/PSM-Standard in SQL1999 eingeführt [ANSI 2003].

¹² vgl. auch <http://www.jcc.com/sql.htm>, 02.12.2006, Part 4 enthält die PSM-Funktionalität.

6.1.6 Typisierte Tabellen und Tabellenhierarchien

Kernstück der objektrelationalen Erweiterung sind sicher die typisierten Tabellen oder auch Objekttabellen, die auf einem Typ aufsetzen. Die Spalten einer typisierten Tabelle entsprechen dabei genau dem Typ, wobei die erste Spalte immer das OID-Attribut ist.

```
<CREATE typisierte Table Anweisung> ::=  
  CREATE TABLE Tabellenname OF Typname [ UNDER Supertabellenname ]  
    ( [ <OID-Generierung> ]  
    [ , <Spaltenoptionen> ]  
    [ , <Tabellenbedingungen> ] );
```

Die OID-Generierung entspricht der OID-Typspezifikation und hat folgende Varianten:

- REF IS OID SYSTEM GENERATED: Die OID wird vom System generiert.
- REF IS <OID-Spaltennamen> DERIVED: Die OID wird aus den Werten der Spaltennamen abgeleitet.
- REF IS OID USER DERIVED: Die OID wird vom Benutzer vorgegeben.

Die Spaltenoptionen können Optionen enthalten, die mit WITH OPTIONS eingeleitet werden und mit NOT NULL, CHECK(Klausel) und SCOPE in etwa den COLUMN CONSTRAINTS entsprechen. Die Tabellenbedingungen entsprechen den TABLE CONSTRAINTS aus Abschnitt 5.3.2. Typisierte Tabellen, die einem Wurzeltyp entsprechen, heißen Wurzeltabellen. Im einfachsten Fall wird zum Typ eine Tabelle angelegt, wobei auch mehrere Tabellen auf einem Typ aufsetzen können.

Beispiel

```
CREATE TABLE Person OF Person_t REF IS OID SYSTEM GENERATED;
```

Zusätzlich können auch Spaltenoptionen und Tabellenbedingungen angelegt werden.

Beispiel

```
CREATE TABLE Angestellter OF Angestellter_t  
  REF IS OID SYSTEM GENERATED,  
  (Beruf WITH OPTIONS NOT NULL,  
   Gehalt WITH OPTIONS CHECK(Gehalt > 0),  
   UNIQUE (Nachname, Vorname));
```

Die mit WITH OPTIONS eingeleiteten Bedingungen sind die Spaltenoptionen. UNIQUE(...) ist eine Tabellenbedingung, da sie keiner Spalte zugeordnet ist, sondern der kompletten Tabelle.

Im Standard SQL2003 ist auch die Definition von Tabellenhierarchien mit Subtabellen und Supertabellen möglich, was hier nicht weiter behandelt wird.

Beispiel

```
CREATE TABLE Person OF Person_t  
  REF IS OID SYSTEM GENERATED;  
  
CREATE TABLE Angestellter OF Angestellter_t UNDER Person  
  REF IS OID SYSTEM GENERATED,  
  (Beruf WITH OPTIONS NOT NULL,  
   Gehalt WITH OPTIONS CHECK(Gehalt > 0),  
   UNIQUE (Nachname, Vorname));
```

In Bezug auf die Datensätze bedeutet dies, dass jeder Angestellte auch implizit in der Personenrelation enthalten ist.

6.1.7 Datenmanipulation mit INSERT, UPDATE und DELETE

Datenmanipulationen (INSERT, UPDATE und DELETE) geschehen in Tupeltabellen und typisierten Tabellen in ganz ähnlicher Form wie im relationalen SQL. Es müssen zusätzlich die Typkonstruktionsregeln ROW, ARRAY oder MULTISSET oder der selbst definierte Datentyp je nach Tabellendefinition verwendet werden.

Beispiel

```
CREATE TABLE Angestellte  
  (Angstellten_NR NUMERIC,  
   Adresse (ROW(Strasse VARCHAR(20)  
              NR      NUMERIC,  
              PLZ     NUMERIC(5),  
              ORT      VARCHAR(20))),  
   Abteilung REF(AbteilungsTyp) SCOPE Abteilungen,  
   Telefonnummer      VARCHAR(20) ARRAY[2],  
   Hobbys            VARCHAR(20) MULTISSET);  
  
INSERT INTO Angestellte VALUES  
  (4711,  
   ROW('Blumstr', 12, 42897, 'Remscheid'),  
   Abteilungen(2),  
   ARRAY ['0221 827612', '02261 8196 100'],  
   MULTISET(['Lesen', 'Reiten', 'Schwimmen']));
```

Benutzerdefinierte Datentypen werden in ähnlicher Weise verwendet:

```
CREATE TYPE Adress_Typ AS (Strasse VARCHAR(20)  
                           NR      NUMERIC,  
                           PLZ     NUMERIC(5),  
                           ORT      VARCHAR(20));
```

```
CREATE TABLE Person (Nachname VARCHAR(20),  
                    Vorname  VARCHAR(20),  
                    Adresse   Adress_Typ);
```

```
INSERT INTO Person  
VALUES ('Schmidt', 'Horst',  
       Adress_Typ('Blumstr', 12, 42897, 'Remscheid'));
```

Auch bei UPDATE-Anweisungen muss der Typkonstruktor bzw. der selbst definierte Typ mit angegeben werden:

```
UPDATE Person
SET    Adresse = Adress_Type('Waldstr', 24, 42897, 'Remscheid');
```

Die DELETE-Anweisung ist fast identisch mit der relationalen Anweisung aus Abschnitt 5.4.3, denn z.B. bei Tabellenhierarchien werden die Einträge in der kompletten Hierarchie entfernt.

6.1.8 Objektrelationale Anfragen

Die SELECT-Anfrage entspricht der SELECT-Anweisung aus Abschnitt 5.5, insbesondere Abschnitt 5.5.1 und 5.5.11. Zusätzlich muss natürlich festgelegt werden, wie Anfragen auf objektrelationalen typisierten Tabellen und Tupeltabellen aufgebaut sein müssen. Da die Syntax sich hier nicht von Oracle unterscheidet, wird auf Abschnitt 6.1.9 verwiesen.

Neu hinzugekommen sind der CAST-Operator¹³ zur Typumwandlung (vgl. Abschnitt 5.2.2) und das CASE-Konstrukt, neben den Join-Operatoren, die schon im Abschnitt 5.5.4 behandelt wurden. Um Kollektionen in eine Tabelle umzuwandeln, bietet SQL2003 den UNNEST-Operator, der unter Oracle dem Operator TABLE (vgl. Abschnitt 6.1.9) entspricht. Der ONLY-Operator dient dazu, in einer Tabellenhierarchie nur diejenigen Tupel zu selektieren, die nur zum Supertyp, aber nicht zu einem Subtyp gehören. Ausgewählte Subtabellen lassen sich mit EXCEPT CORRESPONDING ausschließen. Die Typen einer Instanz können über IS OF Typnamen selektiert werden¹⁴.

6.1.9 Objektrelationales SQL unter Oracle

Auch dieses Kapitel kann die objektrelationalen Konstrukte von Oracle nur in kleinen Auszügen beschreiben, eine ausführliche Darstellung ist in [Hohenstein 2002] zu finden.

6.1.9.1 Basisdatentypen (LOB und XMLType)

Neben den LOB-Typen BLOB und CLOB gibt es noch NCLOB für lange Texte mit nationalen Zeichensatz und BFILE. BFILE ist ein Zeiger auf ein Large Object, welches nicht in der Datenbank, sondern im Dateisystem außerhalb der Datenbank abgelegt ist. Die Funktionen EMPTY_BLOB(), EMPTY_CLOB() und BFILENAME('Verzeichnis', 'Datei') initialisieren die entsprechenden LOB-Tupel. Oracle LOBS bestehen aus zwei Teilen: den eigentlichen LOB-Daten und dem Zeiger (Locator). Die LOB-Daten können bis zu 4 GByte groß sein und liegen bei kleinen LOBS, die nicht mehr als 4000 Byte besitzen, standardmäßig direkt in der Tabelle. Größere LOB-Daten liegen in eigenen Datenbereichen. Bei Abfragen an LOB-Spalten sind nicht erlaubt: GROUP BY-, ORDER BY- und SELECT DISTINCT-Klauseln. Eine Oracle-Tabelle kann mehrere LOB-Spalten besitzen, aber nur eine RAW-Spalte, dem Vorgängerdatentyp von LOB-Daten. Der Datentyp BOOLEAN ist unter Oracle-SQL nicht implementiert. Mit dem PL/SQL-Paket DBMS_LOB¹⁵ können weitere Funktionen auf LOB-Spalten wie READ, WRITE, APPEND, COPY und andere ausgeführt werden. Oracle bietet umfangreiche Möglichkeiten, um mit XML¹⁶ zu arbeiten und z.B. XML-Dokumente auf ein Datenbankschema abzubilden.

¹³ vgl. [Türker et al. 2006, S. 181]

¹⁴ Einzelheiten findet man wieder in [Türker 2003, Kapitel 4] und [Gulutzan et al. 1999].

¹⁵ vgl. Abschnitt 7.1.2

¹⁶ Eine gute Übersicht über XML bietet [Mintert 2002].

Beispiel

```
CREATE TABLE Person_XML OF XMLTYPE;
INSERT INTO Person_XML VALUES (XMLType('<Person>
<P_Nr>1</P_Nr>
</Person>'));
CREATE TABLE XMLType_Column
(Nr      INTEGER,
Text    XMLTYPE)
XMLTYPE Text STORE AS CLOB;
```

Grundsätzlich sind zwei Speicherungsmethoden nämlich feingranular und grobgranular möglich. Als CLOB wird das gesamte Dokument als LOB grobgranular gespeichert. Dies ist die Default-Einstellung. Ab Version Oracle 9.2 kann das Dokument auch feingranular gespeichert werden:

```
CREATE TABLE XMLType_Column2
(Nr      INTEGER,
Text    XMLTYPE)
XMLTYPE Text STORE AS OBJECT RELATIONAL;
```

Bei einer feingranularen Speicherung kann geprüft werden, ob das Dokument einer XML-Schema-Definition genügt. Der Vorteil der feingranularen Speicherung besteht darin, dass auch einzelne Knoten des XML-Dokuments verändert werden können, anstelle des Austauschs des kompletten Baums.

Der Datentyp XMLTYPE bietet eine ganze Reihe von vordefinierten Funktionen, z.B.

- createXML (VARCHAR2 | CLOB),
- getClobVal(),
- extract (VARCHAR2),
- existsNode(VARCHAR2),

mit denen man XML-Dokumente bearbeiten kann und die z.B. in [Oracle Packages 2005]¹⁷ beschrieben sind. In den so erzeugten XML-Dokumenten kann mit einer integrierten XPATH-Funktion (oder der Oracle-TEXT-Funktionalität) gesucht werden:

```
SELECT *
FROM  Person_XML p
WHERE existsNode(VALUE18(p), '/Person/P_Nr')=1;
```

Eine Vertiefung der Oracle-XML-Themen finden Sie neben der Oracle-Dokumentation [Oracle Packages 2005] auch in [Hohenstein et al. 2003].

6.1.9.2 Objektrelationale Typkonstruktoren und Regeln

Oracle setzt die Typkonstruktoren aus SQL2003 weitgehend um, wie die folgende Abbildung zeigt.¹⁹ Bei den Regeln gibt es einige Einschränkungen, da nicht alle Typkonstruktoren aufeinander angewendet werden können.

¹⁷ vgl. http://download-west.oracle.com/docs/cd/B10501_01/appdev.920/a96620/toc.htm, 29.12.2006

¹⁸ Die Value-Funktion wird im Abschnitt 6.1.9 beschrieben.

¹⁹ Die Grafik ist aus [Türker 2003, S. 128] entnommen.

VARRAY ersetzt hier den SQL-Typkonstruktor ARRAY, also eine geordnete Menge von Elementen des gleichen Typs mit einer fest definierten Anzahl. TABLE unterscheidet sich von VARRAY nur dadurch, dass die Anzahl der Elemente nicht festliegt, und entspricht daher nicht dem MULTISSET-Konstruktur.

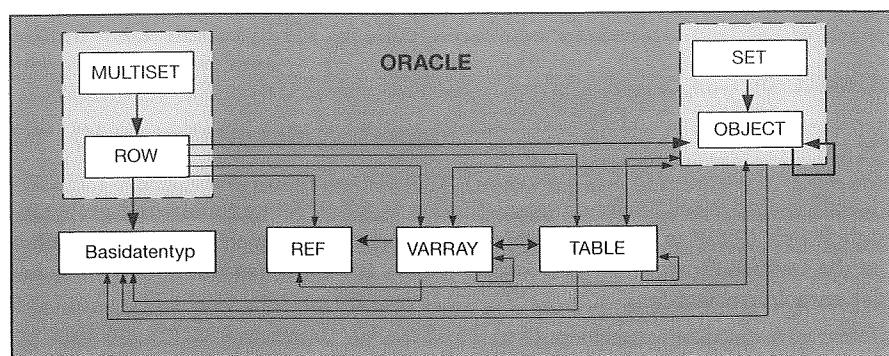


Abbildung 6.3: Typkonstrukte Oracle

6.1.9.3 Strukturierte Typen und Vererbung

Benutzerdefinierte Datentypen können mit wenig veränderter Syntax gegenüber dem Standard-SQL definiert werden. Allerdings unterscheidet Oracle zwischen Objekttypen, VARRAY-Typen, TABLE-Typen und abgeleiteten Subtypen.

Objekttypen	<code>CREATE TYPE AS OBJECT ...</code>
VARRAY-Typen	<code>CREATE TYPE AS VARRAY ...</code>
TABLE-Typen	<code>CREATE TYPE AS TABLE ...</code>
Abgeleitete Typen	<code>CREATE TYPE UNDER <Supertypname></code>

```
<CREATE TYPE Anweisung Object (Auszug20)> ::=  
CREATE OR REPLACE TYPE Typname [UNDER Supertypname ] AS OBJECT  
(<Attributdefinition> [, <Attributdefinition>]... )  
[ [ MEMBER | STATIC ] [ <Methodenspezifikation>]  
[ , [ MEMBER | STATIC ] [ <Methodenspezifikation> ]...] ]...  
[ NOT INSTANTIABLE ]  
[ NOT FINAL ];  
  
<Methodenspezifikation> ::=  
[ PROCEDURE | FUNCTION | Name  
[ ( Parametername <Para_Typ> <Datentyp>  
[ , Parametername <Para_Typ> <Datentyp>]... ) ]  
[ RETURN <Datentyp> ]21  
  
<Para_Typ> ::= /IN | OUT | INOUT/
```

20 Die vollständige Syntax ist in [Hohenstein et al. 2002] enthalten.

21 vgl. Abschnitt 7.1.2, Funktionen haben einen Rückgabewert, Prozeduren nicht.

Vom Standard-SQL unterscheidet sich die Syntax dadurch, dass AS OBJECT vorgeschrieben ist, wenn der Typ nicht von einem anderen Typ erbt. Die Klausel NOT FINAL ist optional, FINAL ist die DEFAULT-Einstellung. Die OID-Generierung wird nicht beim Typ vereinbart, sondern erst bei den Tabellen, die auf den Typ aufsetzen. Man kann einen Typ mit OR REPLACE ersetzen, falls er in der Datenbank schon vorhanden ist. Dabei ist zu beachten, dass ein Typ nur gelöscht werden kann, falls keine Abhängigkeiten zu anderen Typen oder Tabellen mehr bestehen.

Sehr verschieden vom SQL-Standard ist die Syntax, um eine Methode zum Typ zu definieren. Eine Funktion oder Prozedur, die zu einem Typ gehört, ist eine in PL/SQL (vgl. Abschnitt 7.1), Java oder C programmierte Routine²², die entweder eine MEMBER- oder eine STATIC-Routine sein kann. Eine MEMBER-Routine bezieht sich auf eine Instanz, das heißt eine Tabelle, die auf diesem Typ basiert. Eine STATIC-Routine bezieht sich auf den Typ selbst. Eine MEMBER-Methode wird aufgerufen über Instanzname.method(), eine STATIC-Routine über Typname.method(). In der MEMBER-Routine existiert immer ein implizites Argument SELF im Methoden-Body, das die Instanz repräsentiert, für die die Methode aufgerufen wurde. SELF existiert bei STATIC-Routinen natürlich nicht. Ohne LANGUAGE-Option ist immer ein PL/SQL-Programm gemeint, mit der Option kann stattdessen auch ein Java- oder C-Programm benutzt werden.

Die zu einem Typ gehörenden Methoden werden über CREATE TYPE BODY spezifiziert, falls der Typ Methoden besitzt.

```
<CREATE TYPE BODY Anweisung > ::=  
CREATE [ OR REPLACE ] TYPE BODY Typname  
AS [ MEMBER | STATIC ] <Proz_oder_Funktion_Dekl>  
[ , [ MEMBER | STATIC ] <Proz_oder_Funk_Dekl> ] ...  
END;  
  
<Proz_oder_Funktion_Dekl> ::=  
[ PROCEDURE | FUNCTION | Name  
[ ( Parametername <Para_Typ> <Datentyp>  
[ , Parametername <Para_Typ> <Datentyp>]... ) ]  
[ RETURN <Datentyp> ]  
[ IS | AS ] [ <PL/SQL_Block> | LANGUAGE / JAVA | C | <JC_Dek> ]  
  
<PL/SQL_Block> ::= ein PL/SQL-Programm23  
  
<JC_Dek> ::= Deklaration des Java/C-Programms
```

Funktionen haben im Gegensatz zu Prozeduren immer einen Rückgabewert.

Beispiel

```
CREATE OR REPLACE TYPE Adresse_t AS OBJECT  
(Strasse VARCHAR(30),  
NR INTEGER,  
PLZ INTEGER,  
Ort VARCHAR(20),  
LAND VARCHAR(50)) NOT FINAL;
```

22 Routine bedeutet hier: Prozedur oder Funktion, vgl. Abschnitt 7.1.2.

23 vgl. Abschnitt 7.1.2

```

CREATE OR REPLACE TYPE Person_t AS OBJECT
(P_Nr          INTEGER,
Nachname      VARCHAR(50),
Vorname       VARCHAR(50),
Geschlecht    CHAR(1),
Adresse       Adresse_t,
Zeitstempel   DATE) NOT FINAL;

CREATE OR REPLACE TYPE Angestellter_t UNDER Person_t
(Aufgabenbeschreibung CLOB,
Beruf           VARCHAR(10),
Eintrittsdatum  DATE,
Gehalt          NUMERIC(5,2),
Abzuege        NUMERIC(5,2)) NOT FINAL;

CREATE OR REPLACE TYPE Lieferanten_t UNDER Person_t
(Telefonnummer  VARCHAR(30),
Beschreibung   VARCHAR(30));

CREATE OR REPLACE TYPE Kunden_t UNDER Person_t
(Telefonnummer  VARCHAR(30),
Beschreibung   VARCHAR(30));
/
SHOW24 ERRORS

```

Außer den Objekttypen (AS OBJECT) gibt es noch VARRAY-Typen und TABLE-Typen.

```

<CREATE TYPE Anweisung VARRAY> ::= 
  CREATE OR REPLACE TYPE Typname AS VARRAY (<Länge>) OF <Datentyp>;

<CREATE TYPE Anweisung TABLE> ::= 
  CREATE OR REPLACE TYPE Typname AS TABLE OF <Datentyp>;

```

Weder bei den VARRAY-Typen noch bei den Tabellentypen sind Methodendeklarationen zugelassen. Diese Typen heißen Kollektionstypen und haben auch keine FINAL- bzw. INSTANTIABLE-Option. Es kann also weder ein abgeleiteter Typ gebildet noch eine Instanz angelegt werden. Der Datentyp kann entweder ein Basisdatentyp oder ein selbst definierter Datentyp sein, der über CREATE TYPE(..) definiert ist. Die Länge ist eine natürliche Zahl. VARRAYS und TABLE-Typen erlauben lesenden Zugriff auf einzelne Spaltenwerte über einen INDEX, aber keinen UPDATE oder INSERT auf einzelne VARRAY- und TABLE-Werte.

Beispiel

```

CREATE OR REPLACE TYPE Telefon_varray AS VARRAY(3) OF VARCHAR2(20);
CREATE OR REPLACE TYPE Hobbys_table AS TABLE OF VARCHAR2(20);
CREATE OR REPLACE TYPE Adressenliste AS TABLE OF Adresse_t;

```

²⁴ Mit SHOW ERRORS wird unter SQL*PLUS eine Fehlermeldung ausgegeben, falls der Typ Fehler enthält. Der „/“ ist eine Abkürzung für RUN und startet die Anweisung.

Die Informationen, welche Typen mit welchen Attributen in der Datenbank angelegt sind, lassen sich abrufen über:

```

-- 
--Anzeige der Data Dictionary-Informationen (Oracle)
-- 
SELECT * FROM USER_TYPES;
SELECT * FROM USER_TYPE_ATTRS;
SELECT * FROM USER_TYPE_METHODS;

```

6.1.9.4 Typisierte Tabellen und Tabellenhierarchien

Auch unter Oracle kann man unter einem Objekttyp eine typisierte Tabelle definieren, die hier sinnigerweise Objekttabellen heißen und auch in der Syntax erheblich vom Standard abweichen.

```

<CREATE TABLE Objekttabelle> ::= 
  CREATE TABLE Tabellename OF Typname(
  [<SUBSTITUTABLE Klausel>]
  [ Spaltenname <Spaltenbedingung>
  [ , Spaltenname
  <Spaltenbedingung> ]... ]
  [ <Tabellenbedingung> [ , <Tabellenbedingung> ]... ]
  [<OID Generierung>]
  [<Store Klausel>]
  [<Spaltensubstitutionsklausel>]);

<OID Generierung> ::= 
  OBJECT IDENTIFIER IS / SYSTEM GENERATED | PRIMARY KEY /

<Store Klausel> ::= 
  NESTED TABLE <Tabellenwertige Spalte> STORE AS Tabellename

```

Hierbei gilt:

- Typname ist ein über CREATE Type AS OBJECT(...) angelegter benutzerdefinierter Datentyp.
- Mit der SUBSTITUTABLE-Klausel und der Spaltensubstitutionsklausel wird angegeben, ob ein Objekt eines Obertyps durch ein abgeleitetes Objekt eines Subtyps ersetzt werden kann. Nähere Einzelheiten findet man in der Oracle-Dokumentation²⁵ oder bei Hohenstein²⁶. Die SUBSTITUTABLE-Klausel und die Spaltensubstitutionsklausel sind nur relevant, wenn in der CREATE TABLE-Anweisung Objekttypen vorkommen, die von anderen Supertypen abgeleitet sind.
- Die Integritätsbedingungen entsprechen den im Abschnitt 5.3.2. definierten Bedingungen, also Spaltenbedingungen oder Tabellenbedingungen.
- Die OID-Generierung bietet zwei Möglichkeiten: SYSTEM GENERATED bedeutet, dass eine datenbankweit eindeutige OID erzeugt wird, PRIMARY KEY entspricht dem Konzept eines Primärschlüssels aus Kapitel 5. Eine benutzergenerierte OID wie unter Standard-SQL gibt es nicht.

²⁵ vgl. [Oracle SQL 2005]

²⁶ vgl. [Hohenstein et al. 2002, S. 60 ff.]

- Mit der STORE-Klausel können tabellenwertige Spalten beschrieben werden. Für jede Spalte mit einem benutzerdefinierten tabellenwertigen Kollektionstyp (VARRAY, TABLE-Type) muss eine Store-Klausel definiert werden, die angibt, in welcher inneren Tabelle die tabellenwertigen Spalten abgelegt werden.
- Methodendefinitionen sind nur in der zugehörigen CREATE TYPE-Anweisung vorgesehen, bei objektrelationalen CREATE TABLE-Anweisungen gibt es sie nicht.

Beispiel

```
CREATE Table Person OF Person_t OBJECT IDENTIFIER IS SYSTEM GENERATED;
```

Zusätzlich können auch Integritätsbedingungen angelegt werden.

Beispiel

```
CREATE TABLE Angestellte OF Angestellter_t
(Beruf      NOT NULL,
Gehalt     CHECK(Gehalt > 0),
UNIQUE (Nachname, Vorname))
OBJECT IDENTIFIER IS SYSTEM GENERATED;
```

Unter Oracle-SQL kann auch eine geschachtelte Tabelle, also ein tabellenwertiges Attribut, definiert werden. Dies geschieht über NESTED TABLE und die STORE AS-Klausel für jede Spalte, die zu einem Kollektionstyp gehört.

Beispiel

```
CREATE OR REPLACE TYPE Ansprechpartner_t AS OBJECT
(Nachname VARCHAR2(80),
Vorname  VARCHAR2(50),
Telefon   VARCHAR2(20));
CREATE OR REPLACE TYPE Anspr_table AS TABLE OF Ansprechpartner_t;
CREATE TABLE Kunden2
(Firmenname  VARCHAR2(100),
Adresse    Adresse_t,
Ansprechpartner Anspr_table);
NESTED TABLE Ansprechpartner STORE AS Ansprechpartner_table;
```

Zum Schluss dieses Kapitels folgt noch ein Beispiel, welches eine Vererbungshierarchie für typisierte Tabellen basierend auf den Typdefinitionen definiert.

Beispiel

```
CREATE TABLE Person OF Person_t
OBJECT IDENTIFIER IS SYSTEM GENERATED;
CREATE TABLE Kunden OF Kunden_t
OBJECT IDENTIFIER IS SYSTEM GENERATED;
CREATE TABLE Lieferanten OF Lieferanten_t
OBJECT IDENTIFIER IS SYSTEM GENERATED;
Die Typen Kunden_t und Lieferanten_t sind vom Supertyp Person_t abgeleitet. Die Tabellen sind insofern unabhängig voneinander, als Kunden und Lieferanten nicht
```

automatisch in der Personentabelle enthalten sind. Objekte vom Typ Person_t können (müssen aber nicht) in den Tabellen Kunden und Lieferanten eingetragen sein, aber nicht umgekehrt. Um einen Zusammenhang zwischen den Daten der Tabellen des Supertyps und den Tabellen der Subtypen zu erzeugen, benötigt man Trigger-Programmierung (vgl. Aufgabe 27 in Kapitel 7).

Es folgt noch ein Beispiel eines Typs mit benutzerdefinierter MEMBER- und STATIC-Methode sowie einer BODY-Deklaration:

```
CREATE OR REPLACE TYPE Kunden_meth_t AS OBJECT
(Firmenname  VARCHAR2(20),
Strasse     VARCHAR2(20),
Hausnummer  VARCHAR2(5),
PLZ         NUMBER(5),
Ort         VARCHAR2(20),
Ansprechpartner Ansprechpartner_t,
MEMBER FUNCTION get_plzort RETURN VARCHAR2,
STATIC FUNCTION get_strnr (p_str IN VARCHAR2,
                           p_hnr IN VARCHAR2)
                           RETURN VARCHAR2);
```

```
CREATE OR REPLACE TYPE BODY Kunden_Meth_t AS
BEGIN
  MEMBER FUNCTION Get_Plzort RETURN VARCHAR2 IS
  BEGIN
    RETURN ( 'D ' || SELF.PLZ || ' ' || SELF.Ort );
  END;
  STATIC FUNCTION Get_Strnr (P_Str IN VARCHAR2,
                             P_HNr IN VARCHAR2)
                             RETURN VARCHAR2 IS
  BEGIN
    RETURN ( P_Str || ' ' || P_HNr );
  END;
END;
/
```

Die Methode SELF verweist auf die Objektinstanz, die auf dem Typ basiert, in unserem Beispiel einem Tupel der Tabelle Kunden_Meth:

```
CREATE TABLE Kunden_Meth OF Kunden_Meth_t;
```

6.1.9.5 Tupeltabellen

Tupeltabellen sind Tabellen, die nicht auf einem einzelnen Typ basieren, sondern die Spaltendefinitionen haben, wie auch in traditionellen relationalen Tabellen üblich. Zusätzlich zu den Standarddatentypen können auch selbst definierte Datentypen oder Binary Large Objects (BLOB, CLOB, BFILE) benutzt werden. In der CREATE-Table-Anweisung sind im Gegensatz zum Standard-SQL keinerlei unbenannte Typkonstruktoren zugelassen, bis auf den REF-Konstrukt. Die anderen Konstruktoren müssen vorher mit CREATE TYPE angelegt und damit benannt werden.

Beispiel

```
CREATE OR REPLACE TYPE Telefonliste_t AS VARRAY(3) OF INTEGER;
CREATE TABLE Angestellte3
(Angestellten_Nr   NUMERIC,
Adresse          Adresse_t,
Telefon          Telefonliste_t);
```

Es folgt ein Beispiel zu einem REF-Beziehungstyp unter Oracle

Beispiel

```
CREATE TABLE Angestellte OF Angestellte_t OBJECT
  IDENTIFIER IS PRIMARY KEY;

CREATE TABLE Kunden_ref
  (Firmenname      VARCHAR2(100),
   Adresse         Adresse_t,
   Ansprechpartner Ansprechpartner_t,
   Sachbearbeiter_ref REF Angestellter_t SCOPE IS Angestellte);
```

6.1.9.6 Datenmanipulation mit INSERT, UPDATE und DELETE

Datenmanipulationen (UPDATE, INSERT und DELETE) geschehen in Tupeltabellen und typisierten Tabellen in ganz ähnlicher Form wie im relationalen SQL. Für benutzerdefinierte Datentypen ist immer automatisch ein Konstruktor (im Beispiel Adresse_t) definiert, der den gleichen Namen wie der selbst definierte Datentyp hat.

Beispiel

```
INSERT INTO Angestellte3 VALUES
  (2, 'Müller', 'Willi', 'M',
   Adresse_t('Feldweg', 28, 42115, 'Wuppertal', 'Deutschland'),
   SYSDATE, EMPTY_CLOB(), 'Schlosser', SYSDATE, 800, 400);
```

Auch bei UPDATE-Anweisungen muss der selbst definierte Typ als Konstruktor mit angegeben werden:

```
UPDATE Angestellte3
SET Adresse = Adresse_t('Waldstr', 24, 42897, 'Remscheid', 'Deutschland')
WHERE Angestellten_Nr = 2;
```

Es folgt noch ein Beispiel eines INSERT in eine typisierte Objekttabelle Kunden, die direkt vom Typ Kunden_t abgeleitet ist:

```
INSERT INTO Kunden VALUES (
  2, 'Paul', 'Hugo', 'M',
  Adresse_t('Bahnweg', 34, 42115, 'Wuppertal', 'Deutschland'),
  SYSDATE, '02196 55555', 'Witzbold des Hauses');
```

Und ein INSERT in die geschachtelte Tabelle Kunden2:

```
INSERT INTO Kunden2 VALUES (
  'Bahlsen',
  Adresse_t('Bahnweg', 34, 42115, 'Wuppertal', 'Deutschland'),
  Anspr_table(Ansprechpartner_t('Toller', 'Hecht', '022197777'),
  Ansprechpartner_t('Muller', 'Franz', '0221988888')));
```

Im Gegensatz zu VARARRAYS erlauben geschachtelte Tabellen neben lesendem Zugriff auf einzelne Spaltenwerte auch UPDATES und DELETES auf einzelnen Einträgen.

Als letztes Beispiel folgen noch zwei INSERT-Anweisungen zu einer REF-Beziehung:

-- Einfügen mit Sachbearbeiter-Referenz

```
INSERT INTO Kunden_ref
  SELECT 'Referenz GmbH',
    Adresse_t('Weg', '10', 51643, 'Gummersbach', 'Deutschland'),
    Ansprechpartner_t('Bäcker', 'Emil', '02261567893'),
    REF(m)
  FROM Angestellte m
  WHERE m.Angestellten_Nr = 2;
```

-- Einfügen ohne Sachbearbeiter-Referenz

```
INSERT INTO Kunden_ref(Firmenname, Adresse, Ansprechpartner) VALUES (
  'Ohne_REF AG',
  Adresse_t('Weg', '10', 51643, 'Gummersbach', 'Deutschland'),
  Ansprechpartner_t('Müller', 'Erna', 02261567893));
```

6.1.9.7 Objektrelationale Anfragen und SELECT-Anweisungen

Bei der SELECT-Anweisung gibt es eine etwas unterschiedliche Syntax – je nachdem, ob es sich um eine Tupeltabelle mit benutzerdefiniertem Datentyp handelt oder um eine typisierte Objekttabelle. Bei einer typisierten Tabelle kann man eine VALUE-Funktion benutzen, die den Inhalt der Tabelle als Objekt ausgibt.

Zu beachten ist, dass benutzerdefinierte Spalten sowie einzelne Felder daraus nur mittels Tabellenalias ansprechbar sind.

Beispiel

Tupeltabelle mit benutzerdefiniertem Datentyp Adresse:

```
SELECT * FROM Angestellte;
SELECT Adresse FROM Angestellte;
SELECT a.Adresse.ORT FROM Angestellte a;
```

Beispiel

Typisierte Objekttabelle zum Typ Person_t:

```
SELECT * FROM Person;
SELECT Adresse FROM Person;
SELECT VALUE(p) FROM Person p;
```

Beim letzten SELECT wird der Inhalt der Tabelle Person als Objekt ausgegeben, nicht als einzelner Wert wie bei der vorigen Anweisung.

Beispiel

Anfragen an geschachtelte Tabelle

```
SELECT * FROM Kunden2;
```

Ergebnis:

FIRMEN-NAME	ADRESSE (STRASSE,NR, PLZ, ORT, LAND)	ANSPRECHPARTNER(NACHNAME, VORNAME, TELEFON)
Bahlsen	ADRESSE_T ('Bahnweg', 34, 42115, 'Wuppertal', 'Deutschland')	ANSPR_TABLE (ANSPRECHPARTNER_T('Toller', 'Hecht', '022197777'), ANSPRECHPARTNER_T('Muller', 'Franz', '0221988888'))

Bei der vorigen Anfrage handelte es sich um eine „nested Query“, da die komplette Tabelle mit den eingebetteten Kollektionen angezeigt wird.

```
SELECT k.Firmenname, a.*  
FROM Kunden2 k, TABLE (k.Anprechpartner) a;
```

Ergebnis:

FIRMENNAME	NACHNAME	VORNAME	TELEFON
Bahlsen	Toller	Hecht	022197777
Bahlsen	Muller	Franz	0221988888

Bei der zweiten Anfrage handelt es sich um eine „unnested Query“, da die enthaltene geschachtelte Tabelle mit dem Operator TABLE flachgeklopft und auf eine relationale Tabelle abgebildet wird. Die Benutzung eines Tabellenalias (k und a) ist bei geschachtelten Tabellen obligatorisch.

```
SELECT k.Firmenname, CURSOR (SELECT * FROM TABLE (k.Anprechpartner))  
FROM Kunden2 k;
```

Ergebnis:

FIRMENNAME	CURSOR(SELECT * FROM TABLE (k.Anprechpartner))
Bahlsen	CURSOR STATEMENT: 2

Bei der letzten Abfrage wird die Anzahl der Einträge in der geschachtelten Tabelle Anprechpartner gezählt.

Die REF-Beziehung ersetzt einen Fremdschlüsselverweis wie im relationalen Modell üblich. Es wird ein Zeiger auf das Referenzobjekt in der Master-Tabelle angelegt. Erstaunlicherweise wird hier zugelassen, dass Zeiger ins Leere verweisen, wenn das Master-Objekt gelöscht wird. Es können also sogenannte „Dangling-Tupel“ entstehen, die sich auch mit IS [NOT] DANTLING abfragen lassen.

6.1.9.8 Anfragen und INSERTS zur REF-Beziehung

```
-- Anzeige mit REF-Pointer  
SELECT * FROM Kunden_ref;  
  
-- Anzeige mit Sachbearbeiternamen  
SELECT k.Firmenname, k.Sachbearbeiter_ref.Nachname  
FROM Kunden_ref k;  
  
-- Dangling Tuples  
SELECT k.Firmenname, k.Sachbearbeiter_ref.Nachname
```

```
FROM Kunden_ref k  
WHERE k.Sachbearbeiter_ref IS NOT DANTLING;  
SELECT k.Firmenname, k.Sachbearbeiter_ref.Nachname  
FROM Kunden_ref k  
WHERE k.Sachbearbeiter_ref IS DANTLING;  
COMMIT;  
-- Löscht referenziertes Objekt (Master)  
DELETE FROM Angestellte3 WHERE Angestellten_Nr = 2;  
SELECT k.Firmenname, k.Sachbearbeiter_ref.Nachname  
FROM Kunden_ref k  
WHERE k.Sachbearbeiter_ref IS DANTLING;  
ROLLBACK;
```

Den Abschluss dieses Kapitels bilden einige INSERT- und SELECT-Anweisungen in die Tabelle Kunden_Meth, die eine selbst definierte MEMBER-Funktion und eine STATIC-Funktion hat.

6.1.9.9 INSERT und SELECT zu einer typisierten Tabelle mit Methodenaufruf

```
INSERT INTO Kunden_Meth  
VALUES ('Konkurs AG', 'Blumenweg', '14', 51647, 'GM',  
       Anprechpartner_t('Müller', 'Elfriede', 02261819623));  
  
INSERT INTO Kunden_Meth  
VALUES ('Pleite GbR.', 'Baumalle', '18c', 51099, 'Köln',  
       Anprechpartner_t('Mende', 'Else', 02211665677));  
  
SELECT k.get_plzort() PLZ_Ort,  
      Kunden_Meth_t.Get_Strnr(k.Strasse, k.Hausnummer) Str_HNr  
FROM Kunden_Meth k;
```

Die MEMBER-Methode get_plzort wird für jedes Tupel, das in die Tabelle Kunden_meth eingefügt wurde, aufgerufen. Die STATIC-Methode get_strn wird mit dem Typnamen Kunden_meth_t in Punktnotation aufgerufen. Die Methoden enthalten eine Klammerung (), auch wenn, wie hier bei der Methode get_plzort(), keine Parameter notwendig sind.

6.1.10 Objektrelationales SQL unter MySQL

MySQL ist noch nicht so weit, objektrelationale Konstrukte einzubauen, mit Ausnahme der Aufzählungstypen ENUM, SET sowie des Large Objects, BLOB.

Beispiel

```
CREATE TABLE Angestellte3  
(Angestellten_Nr   NUMERIC,  
  Nachname        VARCHAR(20),  
  Vorname         VARCHAR(10),  
  Geschlecht      ENUM('W', 'M'),  
  Bild            BLOB,  
  Telefonliste    SET());
```

Das ENUM-Feld „Geschlecht“ kann einen der Werte „W“ oder „M“ aufnehmen und ersetzt damit einen CHECK CONSTRAINT.²⁷ Der SET-Typkonstruktor entspricht dem Typkonstruktor ARRAY aus dem Standard SQL2003, wobei die Länge des Arrays immer 64 ist und die Felder aus Strings bestehen.

MySQL bietet für Binary Large Objects acht Datentypen an, die sich in der Größe und vom Typ (BLOB oder TEXT) unterscheiden:

- TINYBLOB und TINYTEXT haben maximal 255 Byte.
- BLOB und TEXT haben maximal 64 Kbyte.
- MEDIUMBLOB und MEDIUMTEXT haben maximal 16 Mbyte.
- LONGBLOB und LONGTEXT haben maximal 2 Gbyte.

Die BLOB-Typen sind für Bilder, Videos etc. gedacht, die TEXT-Typen für lange Texte anstelle der CLOBS in Standard-SQL. Allerdings können die Daten aus BLOB-Spalten nur als ganzes Objekt aus der Datenbank geladen werden und es stehen keine Funktionen zur Verfügung, um mit den BLOB-Spalten in der Datenbank zu arbeiten, was das Abspeichern von BLOB-Dateien großen Umfangs erschwert. Darüber hinaus können die MySQL-Server nur Pakete verarbeiten, die höchstens 1 Gbyte groß sind, wenn man die Einstellung des Parameters max_allowed_packet nicht höher setzt. Insgesamt wird von MySQL²⁸ selbst empfohlen, diese Datentypen (BLOB, TEXT) in der Regel nicht anzuwenden, sondern stattdessen nur einen Zeiger auf das Dateisystem, das die Binärdaten enthält, in der Datenbank zu speichern.

6.1.11 Objektrelationale Abbildung

In vielen Fällen wird eine objektrelationale Analyse betrieben, da sie die reale Welt besser beschreibt, während die Implementierung noch auf traditionelle, relationale Weise in SQL geschieht. In diesem Kapitel zeigen wir, wie man

- OIDS (Object Identifier) auf relationalen Datenbanken abbildet,
- Kollektionen (ARRAY, MULTISSET) relational beschreibt,
- Vererbungshierarchien auf relationale Datenbanken abbildet.

Diese Abbildung bezeichnet man auch als objektrelationale Abbildung. Eine ausführliche Darstellung findet man z.B. in Türker²⁹ und Elmasri.³⁰ Wir haben hier eine Auswahl getroffen und demonstrieren die verschiedenen Abbildungsvorschriften anhand von Beispielen.

6.1.11.1 Abbildung von OID-Spalten auf Primärschlüssel

Eine OID stellt in objektrelationalen Systemen eine systemweit eindeutige ID dar, die auch dereferenziert werden kann und die einmal vergeben für ein Objekt unveränderbar ist. Die OID soll vom System generiert werden.

Naheliegend ist es, solch eine OID durch einen Primärschlüssel nachzubilden. Die automatische Generierung der OID kann man relational einer SEQUENCE (vgl. Abschnitt

5.3.6.) übertragen. Um außerdem zu verhindern, dass ein Primärschlüssel nachträglich verändert wird, kann man einen UPDATE-TRIGGER (Zeilentrigger) in der Datenbank ablegen, der Änderungen am Primärschlüssel zurückweist (vgl. Kapitel 7, Aufgabe 28).

6.1.11.2 Abbildung von Kollektionen auf relationale Konstrukte

In Standard-SQL handelt es sich hier um die Typkonstrukturen MULTISSET und ARRAY, die eine gewollte Verletzung der ersten Normalform erlauben. Der Unterschied besteht darin, dass MULTISSET keine feste Länge hat und auch unterschiedliche Datentypen zugelassen sind.

Zur Umsetzung von Kollektionsspalten in relationale Tabellen gibt es zwei Ansätze:

- Abbildung auf zusätzliche Tabellen (Kollektionstabellen)
- Abbildung auf zusätzliche Spalten (Kollektionsspalten)

Abbildung von Kollektionen auf Kollektionstabellen

Für jedes mengenwertige Attribut wird eine eigene Tabelle gebildet, die über eine Fremdschlüsselbeziehung mit der Master-Tabelle verbunden ist. Das ist sowohl beim MULTISSET-Konstruktor als auch beim ARRAY-Konstruktur möglich.

Beispiel

```
-- MULTISSET-Konstruktor in SQL2003
CREATE TABLE Angestellte
  (Angestellten_Nr      NUMERIC PRIMARY KEY,
   Nachname            VARCHAR(20),
   Vorname             VARCHAR(10),
   Telefonliste        VARCHAR(20) MULTISSET);

-- Relationale Umsetzung als Kollektionstabelle Telefonliste
CREATE TABLE Angestellte
  (Angestellten_Nr      NUMERIC PRIMARY KEY,
   Nachname            VARCHAR(20),
   Vorname             VARCHAR(10));

CREATE TABLE Telefonliste
  (Angestellten_NR     NUMERIC REFERENCES Angestellte(Angestellten_Nr),
   Telefon_Nr          VARCHAR(20),
   PRIMARY KEY          (Angestellten_Nr, Telefon_Nr));
```

Abbildung von Kollektionen auf Tabellen mit Kollektionsspalten

Bei diesem Ansatz muss die Anzahl der Kollektionsspalten festliegen, es muss sich also um ein ARRAY fester Länge handeln. Für jeden Eintrag in die Kollektionsspalte wird dann eine eigene Spalte gebildet.

Beispiel

```
-- ARRAY-Konstruktor in SQL2003
CREATE TABLE Angestellte
  (Angestellten_Nr      NUMERIC PRIMARY KEY,
   Nachname            VARCHAR(20),
   Vorname             VARCHAR(10),
   Telefonliste        VARCHAR(20) ARRAY[3]);
```

²⁷ vgl. Abschnitte 5.3.2 und 5.3.3

²⁸ vgl. [MySQL 2006]

²⁹ vgl. [Türker 2003]

³⁰ vgl. [Elmasri 2003]

```
-- Relationale Umsetzung als Tabelle mit Kollektionsspalten
CREATE TABLE Angestellte
(
    Angestellten_Nr     NUMERIC,
    Nachname           VARCHAR(20),
    Vorname            VARCHAR(10),
    Telefon_Nr1         VARCHAR(20),
    Telefon_Nr2         VARCHAR(20),
    Telefon_Nr3         VARCHAR(20));

```

6.1.11.3 Abbildung von Vererbungshierarchien auf relationale Konstrukte

Diese Abbildung wurde schon im Abschnitt 4.4.2 als Transformation eines EERM auf ein relationales Datenmodell beschrieben (virtuelle, vertikale und horizontale Fragmentierung). Das dort entstandene relationale Datenmodell kann ohne weitere Transformation auf relationale CREATE TABLE-Befehle abgebildet werden. TRIGGER, die die Einhaltung der entsprechenden Integritätsregeln garantieren, findet man in den Aufgaben zu Kapitel 7 (vgl. Kapitel 7, Aufgaben 31 und 32).

Insgesamt hat man bei der relationalen Umsetzung eines objektorientierten Modells eine aufwändige Datendefinition, die noch durch TRIGGER unterstützt werden muss, um die Konsistenz zu gewährleisten. Die SELECT-Abfragen erstrecken sich naturgemäß über mehrere Tabellen, auf die die Daten verteilt sind, die eigentlich logischerweise zusammengehören. Das kann zu Performance-Verlusten führen.

6.2 Objektrationale Anwendungsprogrammierung: JDBC und SQLJ

6.2.1 JDBC

Java feierte in den vergangenen Jahren als Programmiersprache einen Siegeszug ohne gleichen, da diese Sprache objektorientiert, modern und von Anfang an mit dem Internet verbunden war. Java und SQL haben sich dabei neben der MySQL/PHP-Schiene für kleinere Webanwendungen zu einem Quasi-Standard von webbasierten Anwendungen mit Datenbankanbindung entwickelt. Historisch gesehen entstanden diese Kopplungen von Java und SQL aus dem SQL/CLI-Standard (Call Level Interface). Dieses Kapitel befasst sich daher mit CLI-Grundlagen und JDBC, da sich JDBC (und auch SQLJ) am CLI-Standard orientiert.

6.2.1.1 SQL/CLI – ein Standardisierungsvorhaben

CLI-Anwendungen sind ein Beispiel für die sogenannte FAT-Client/THIN-Server-Architektur. Die Anwendungslogik ist auf dem Client lokal gespeichert und die Datenbank befindet sich auf dem Server.

CLI ist eine standardisierte Schnittstelle zur Kommunikation mit Datenbanken ohne Precompiler. CLI gehört zum SQL-Standard seit 1995. Die CLI-API³¹ erlaubt das Erstellen und Ausführen von SQL-Anweisungen zur Laufzeit. Seinen Ursprung hat

³¹ vgl. <http://www.jcc.com/sql.htm>, 01.12. 2006. Auf dieser Seite werden die verschiedenen SQL-Standards beschrieben, in Part 3 ist CLI dargestellt.

CLI in Aktivitäten der SQL Access Group (SAG) mit dem Ziel, einen vereinheitlichten Zugriff auf Datenbanken bereitzustellen.

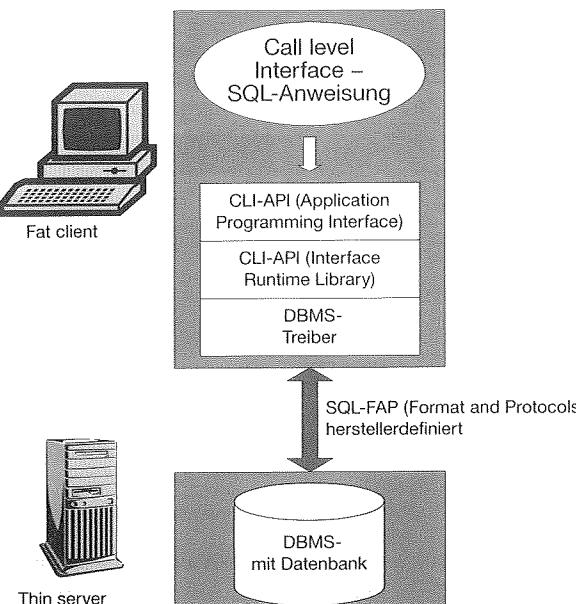


Abbildung 6.4: CLI

Die einzelnen Bestandteile von CLI sind:

- DB-Verbindungsaunahme und Beenden der Verbindung
- Vorbereitung und Ausführen von SQL-Anweisungen („prepare“ und „execute“)
- Binden von Parametern an SQL-Anweisungen („bind“)
- Entgegennahme von Resultaten
- Festlegung von Datentypen über spezielle Codes für jeden Datentyp

Ein CLI-Call wird durch einen proprietären DBMS-Treiber in einen herstellerspezifischen DB-Zugriff umgesetzt. Damit kann prinzipiell jede Datenquelle mit dem geeigneten Treiber angesprochen werden.

Neben standardisiertem SQL/CLI haben sich noch weitere Dialekte entwickelt:

- ODBC (Open Database Connectivity): Microsofts CLI-Variante
 - Beinhaltet die meisten (nicht alle) API-Calls des CLI-Standards
 - Zusätzlich: spezielle Calls zur Unterstützung von Microsoft-Programmen (Access, Excel etc.)
 - Proprietäre CLIs der DB-Hersteller („Native API“)
 - z.B. Oracle Call Interface (OCI)

Diese Dialekte werden hier nicht behandelt. Sie haben einen ähnlichen Aufbau wie die anderen CLI-Bibliotheken.

Mit JDBC wurde die SQL/CLI-Schnittstelle zu einer objektorientierten Schnittstelle weiterentwickelt. Die Architektur ist ähnlich wie bei SQL/CLI.

6.2.1.2 Was ist JDBC?

JDBC (Java Database Connectivity) ist ein Java-API (Application Programming Interface) zur Ausführung von SQL-Anweisungen. Es besteht aus einer Menge von Klassen und Schnittstellen, die in der Programmiersprache Java geschrieben sind. JDBC bietet ein Standard-API für Entwickler von Datenbankwerkzeugen und ermöglicht das Schreiben von Datenbankanwendungen unter Verwendung einer puren Java-Schnittstelle.

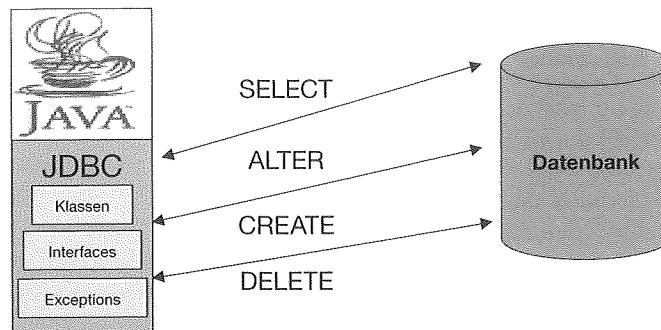


Abbildung 6.5: JDBC

Durch Verwendung von JDBC ist es einfach, SQL-Anweisungen an nahezu jede relationale Datenbank zu senden, d.h., mit dem JDBC-API ist es nicht notwendig, ein Programm für den Zugriff auf eine Oracle-Datenbank zu schreiben, ein anderes Programm für den Zugriff auf eine Informix-Datenbank usw. Mit einer in Java geschriebenen Anwendung muss man sich auch nicht um das Schreiben verschiedener Anwendungen auf verschiedenen Plattformen kümmern. Die Kombination von Java und JDBC ermöglicht es dem Programmierer, das Programm einmal zu schreiben und es überall laufen zu lassen.

Zum Beispiel ist es mithilfe von Java und des JDBC-API möglich, eine Webseite mit einem Applet zu veröffentlichen, das Informationen aus einer entfernten Datenbank verwendet. Oder ein Unternehmen kann JDBC einsetzen, um alle seine Angestellten an eine oder mehrere interne Datenbanken über ein Intranet anzuschließen (selbst wenn im Computernetzwerk der Firma verschiedene Systeme wie z.B. Unix-, Windows-, Macintosh-Rechner vorhanden sind).

JDBC wurde 1996 von SUN³² veröffentlicht und ist seit dem JDK 1.1 in diesem Archiv enthalten. Die aktuelle (2006) Version ist J2SE5.0 mit der API JDBC 3.0. In Java-Programmen wird die JDBC-Schnittstelle mit `import java.sql.*` und `import javax.sql.*` importiert, je nach benötigter Funktionalität.

6.2.1.3 Überlegungen zur Systemarchitektur von JDBC

Eine grundlegende Einführung in JDBC bietet [White et al. 2000], die auch deshalb besonders zu empfehlen ist, da die Autoren maßgeblich an der Entwicklung von JDBC beteiligt waren. Sie unterscheiden vier verschiedene Treibertypen:

Treiber vom Typ 1: JDBC-ODBC-Bridge Bei diesem Treibertyp greift der JDBC-Treiber über die ODBC-Schnittstelle auf die Datenbank zu. Natürlich muss dazu auf jedem Client ein ODBC-Treiber installiert sein. Dieser Treibertyp bietet sich daher hauptsächlich in lokalen Netzwerken an, bei denen auf dem einzelnen Client leicht ODBC installiert werden kann. Außerdem ist durch den Zwischenschritt ODBC der Zugriff verhältnismäßig langsam.

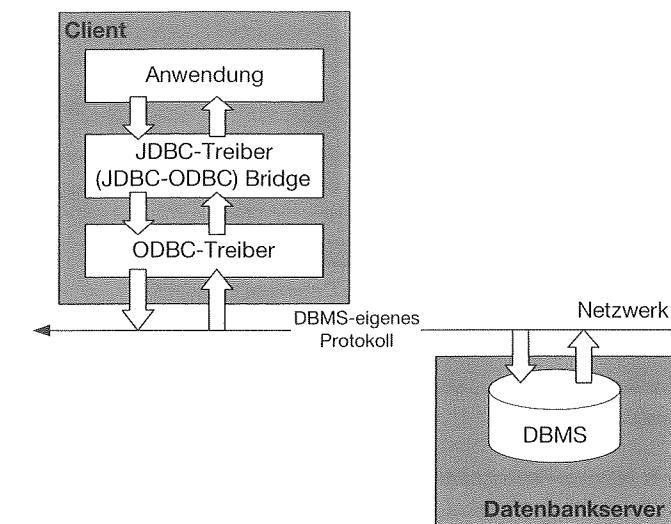


Abbildung 6.6: JDBC-Treiber vom Typ 1

Treiber vom Typ 2: Native-API partly-Java-Treiber Bei diesem Ansatz werden die JDBC-Aufrufe in ein datenbankspezifisches API auf dem Client weitergegeben. Daher sind auch bei diesem Typ zusätzliche Installationen auf dem Client notwendig.

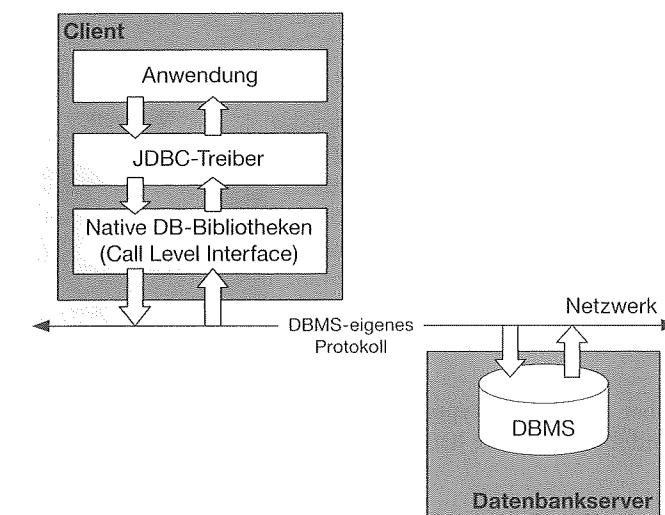


Abbildung 6.7: JDBC-Treiber vom Typ 2

32 vgl. <http://java.sun.com/products/jdbc/>, 10.12.2006

Treiber vom Typ 3: JDBC-Net pure Java-Treiber Treiber vom Typ 3 übersetzen die JDBC-Aufrufe in ein vom DBMS unabhängiges Netzprotokoll und benutzen einen Anwendungsserver. Der Server übersetzt die Aufrufe in das jeweilige DBMS-Protokoll und über die CLI-Schnittstelle wird auf die Datenbank zugegriffen. Bei Treibern vom Typ 3 ist keine zusätzliche Client-Installation notwendig, aber ein Anwendungsserver.

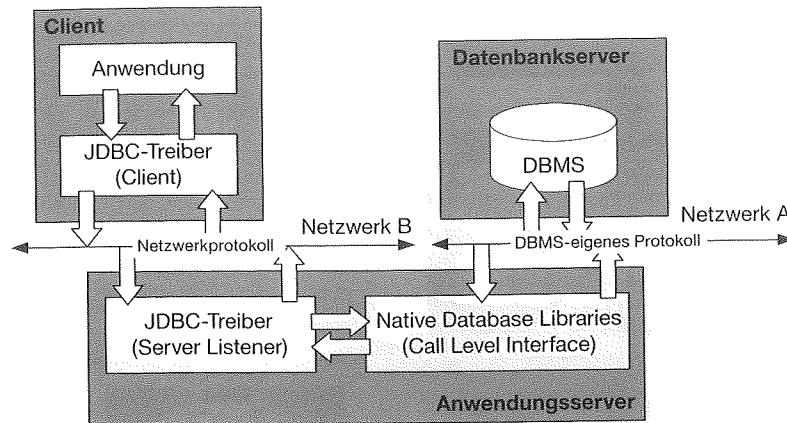


Abbildung 6.8: JDBC-Treiber vom Typ 3

Treiber vom Typ 4: Native-Protocol pure Java-Treiber Bei diesem Treibertyp werden die JDBC-Aufrufe in ein datenbankeigenes Protokoll übersetzt, welches über das Netzwerk direkt auf den Datenbankserver zugreift. Es ist auch keine zusätzliche Client-Installation notwendig.

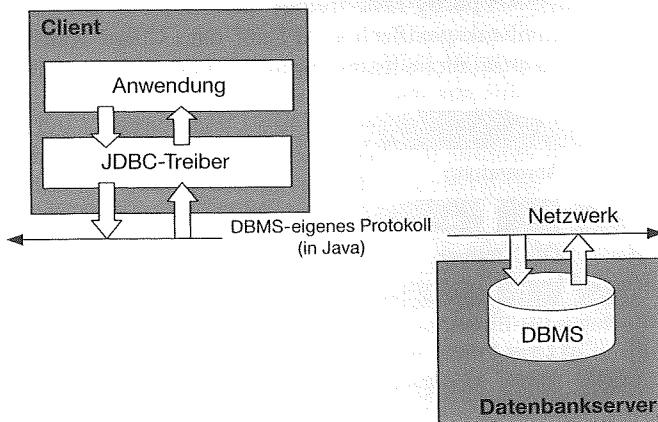


Abbildung 6.9: JDBC-Treiber vom Typ 4

Informationen über alle Treibertypen, eine Treiberbibliothek und viele zusätzliche Informationen findet man auf der JDBC-Seite³³ von Sun.

33 vgl. <http://java.sun.com/products/jdbc/index.html>, 02.12.2006.

6.2.1.4 Sieben Schritte bei der Datenbankanbindung mit JDBC

Bei Datenbankabfragen mittels JDBC gibt es sieben Standardschritte:

- 1 Laden des JDBC-Treibers
- 2 Definition der URL der Datenbankverbindung
- 3 Die Verbindung zur Datenbank herstellen
- 4 Eine Anweisung in Gestalt eines Statement-Objekts erzeugen
- 5 Eine Abfrage oder eine Datenänderung ausführen
- 6 Die Ergebnisse einer Abfrage verarbeiten
- 7 Das Schließen der Datenbankverbindung

Diesen sieben Schritten ist der folgende Abschnitt gewidmet.

1. Schritt: Laden des JDBC-Treibers

Der JDBC-Treiber entstammt einer Klassenbibliothek³⁴, die die Verbindung mit dem aktuellen Datenbankserver herstellt. Um die Klasse zu laden und beim DriverManager zu registrieren, wird die Methode Class.forName verwendet. Der Pfad, in dem sich die Klassenbibliotheken befinden, muss der Java-Virtuell-Maschine als Umgebungsvariable oder direkt beim Aufruf von javac bekanntgegeben werden. Damit kann man eine Klasse laden, ohne eine Instanz zu erzeugen, und eine Klasse referenzieren, deren Name noch nicht bekannt ist. Der Aufruf wird im Allgemeinen in einen try/catch-Block verpackt:

```

try {
    Class.forName("oracle.jdbc.driver.OracleDriver");
    Class.forName("com.sybase.jdbc.SybDriver");
    Class.forName("com.mysql.jdbc.Driver").newInstance();
    Class.forName("connect.microsoft.MicrosoftDriver");
} catch ClassNotFoundException e {
    System.err.println("Fehler beim Laden des DB - Treibers: " + e);
}

```

Hat man einen Treiber geladen, ist er für den Verbindungsaufbau zu einem DBMS verfügbar.

2. Schritt: Definition der URL zu der Datenbankverbindung

Als Nächstes wird der Standort des Datenbankservers angegeben. Dazu wird eine URL eines speziellen Typs verwendet. Die JDBC-URL besteht aus den Komponenten jdbc:<Subprotokoll>:<Subname>. Das Subprotokoll enthält den Namen des zu verwendenden JDBC-Treibers, der Subname die Identifikation der Datenbank mit eventueller Parameterzeichenfolge. Für eine Personal Oracle-Datenbank lautet die URL folgendermaßen:

"jdbc:oracle:oci8:@"

Und für einen Oracle-Datenbankserver:

"jdbc:oracle:thin:@<host>:<port>:<sid³⁵" ;

34 Die JDBC-Klassen müssen in die Java-Umgebung eingebunden werden und heißen unter Oracle classes111.zip bzw. ojdbc.zip.

35 Eine „sid“ ist eine Oracle-Instanz einer Datenbank.

Dementsprechend auch für einen MySQL-Server:

```
"jdbc:mysql://<host>:<port>/mysql"
```

Weitere Beispiele sind:

```
String OracleURL = "jdbc:oracle:thin:scott/tiger@oras1.gm.fh-koeln.de:1521:oras1"
```

```
String MySQLURL = "jdbc:mysql://localhost:3306/mysql";
```

JDBC-Verbindungen werden von den unterschiedlichsten Java-Applikationen, Servlets oder Applets genutzt. Bei Applets ist zu beachten, dass aus Sicherheitsgründen Applets nur Netzwerkverbindungen mit dem Server gestatten, von dem sie geladen wurden.

3. Schritt: Verbindung zur Datenbank herstellen

Hierzu wird die Methode getConnection der Klasse DriverManager benutzt, die als Parameter eine Datenbank-URL, einen Benutzernamen und ein Passwort benötigt.

```
Connection conn = DriverManager.getConnection(url,user,pwd)
```

oder bei einer JDBC-ODBC-Bridge

```
Connection conn = DriverManager.getConnection("jdbc:odbc:<DNS-NAME>")
```

Hier können schon Datenbankinformationen über die Methode getMetaData des connection-Objekts abgefragt werden.

Beispiel

```
DatabaseMetaData dbMetaData = connection.getMetaData();
String productName = dbMetaData.getDatabaseProductName();
System.out.println("Datenbank: " + productName);
```

4. Schritt: eine Anweisung in Gestalt eines Statement-Objekts erzeugen

Ein Statement-Objekt hat die Aufgabe, Anfragen und Datenbankänderungen durchzuführen. Es wird mit der Methode createStatement aus dem Connection-Objekt erzeugt:

```
Statement v_statement = conn.createStatement();
```

Andere Methoden des connection-Objekts, die hier verwendet werden können, sind prepareStatement(), commit(), rollback() und close(), die später behandelt werden.

5. Schritt: eine Abfrage oder eine Datenänderung ausführen

Die Methode executeQuery des Statement-Objekts führt Abfragen durch, die über einen String übergeben werden:

```
String query = "SELECT * FROM Laender";
ResultSet v_resultset = statement.executeQuery(query);
```

Statt executeQuery wird bei Datenbankänderungen die Methode executeUpdate verwendet, wie im Beispiel auf der nächsten Seite deutlich wird. An diese Methode können dann die SQL-Anweisungen INSERT, UPDATE, DELETE oder CREATE übergeben werden. Die einzelnen Strings, die eine SQL-Anweisung enthalten, dürfen nicht mit Semikolon abgeschlossen werden.

6. Schritt: Ergebnisse einer Abfrage verarbeiten

Die Klasse ResultSet bietet eine Vielzahl von Methoden, um in der Ergebnismenge zu navigieren und sie auszugeben. Am einfachsten ist es, das Result-Set mit der Methode next() Zeile für Zeile durchzugehen. Später werden wir noch weitere Einzelheiten der Klasse ResultSet behandeln.

```
while (v_resultset.next()) {
    String sl = v_resultset.getString(1);
    System.out.println(sl);
}
```

Dabei ist zu beachten, dass in einer ResultSet-Zeile der erste Index eine 1 und nicht eine 0 ist. Mit getString bzw. getXXX stehen diverse Methoden zur Verfügung, die die SQL-Datentypen auf Java-Datentypen abbilden.

7. Schritt: Schließen der Datenbankverbindung

Das Statement-Objekt und das connection-Objekt werden geschlossen:

```
v_statement.close();
v_connection.close();
```

Beispiel

aus den Schritten eins bis sieben

```
import java.sql.*; // hier werden die JDBC-Klassen importiert
public class Laender {
    public static void main(String args[]) {
        String url = "jdbc:oracle:thin:@host:1521:dbname";
        Connection v_connection;
        Statement v_statement;
        try {
            Class.forName("oracle.jdbc.driver.OracleDriver");
        } catch(java.lang.ClassNotFoundException e) {
            System.err.print("ClassNotFoundException: ");
            System.err.println(e.getMessage());
        }
        try {
            v_connection = DriverManager.getConnection(url, "user", "passwd");
            v_statement = v_connection.createStatement();
            //drop-Table-Anweisung ausführen
            v_statement.executeUpdate(" DROP TABLE Laender ");
            //CreateTable-Anweisung ausführen
            v_statement.executeUpdate(
                " CREATE TABLE Laender (land_name varchar2(100))");
            // Insert-Anweisung ausführen
            v_statement.executeUpdate(
                " INSERT INTO Laender VALUES('Deutschland')");
            v_statement.executeUpdate(
                " INSERT INTO Laender VALUES('Holland')");
        }
    }
}
```

```

// SELECT-Anweisung
ResultSet v_resultset = v_statement.executeQuery(
    ("SELECT * FROM Laender");
System.out.println("Laender");
while (v_resultset.next()) {
    String s1 = v_resultset.getString(1);
    System.out.println(s1);
}
// Statement und Verbindung schließen
v_statement.close();
v_connection.close();
} catch(SQLException ex) {
    System.err.println("SQLException: " + ex.getMessage());
}
}

```

6.2.1.5 Statement-Arten in JDBC

Im Allgemeinen sieht der Ablauf einer JDBC-Verbindung folgendermaßen aus: Der DriverManager baut über getConnection() eine Verbindung auf, über createStatement wird ein Objekt der Klasse Statement erzeugt. Die Methode executeQuery erzeugt dann ein ResultSet, in dem die Ergebnisse der Abfrage gespeichert sind. Neben der createStatement-Methode gibt es die Methoden prepareStatement und prepareCall des Objekts „connection“. Diese Methoden erzeugen verschiedene Statement-Arten aus JDBC:

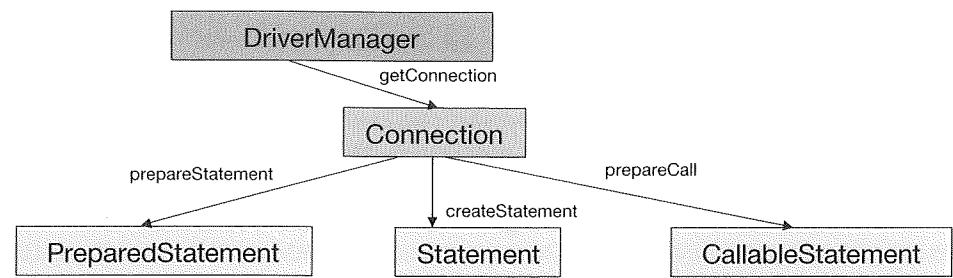


Abbildung 6.10: JDBC-Objekte

Eigenschaften des Statement-Objekts Das Statement-Objekt wird zum Erzeugen statischer SQL-Befehle verwendet. Diesem Objekt können keine Parameter übergeben werden. Ein Beispiel zur Verwendung des Statement-Objekts findet man im vorigen Kapitel.

Eigenschaften des PreparedStatement-Objekts Ein PreparedStatement kann im Unterschied zu einem Statement-Objekt Parameter aufnehmen. Es erbt Methoden und Eigenschaften des Statement-Objekts. Der Platzhalter für Parameter in der SQL-Schablone ist ein Fragezeichen. Durch Vorübersetzung wird die SQL-Syntax schon zur Übersetzungszeit geprüft und ein Performance-Gewinn erzielt. Der Ablauf sieht folgendermaßen aus: Sie definieren ein PreparedStatement-Objekt über

PreparedStatement updateOrt;

Das Objekt wird mit einem String initialisiert, der auch den Platzhalter „?“ für einen oder mehrere Parameter enthalten kann:

```

String updateString = "UPDATE Dozent SET ort = ?";
updateOrt = v_connection.prepareStatement(updateString);

```

Das Objekt updateOrt wird über setXXX (setString, setBoolean, setInt, setShort, setBlob, setFloat, setDate, setNull ...) mit Werten gefüllt, wobei die Position des Werts durch die Spaltennummer bestimmt ist.

```
updateOrt.setString(1, "Berlin");
```

Zum Schluss wird der Vorgang mit der Methode executeUpdate() ausgeführt.

Ein Beispiel zur Verwendung eines PreparedStatement zeigt der folgende Java-Code:

Beispiel

```

import java.sql.*;
public class preparedStatement {
    public static void main(String[] args) {
        String url = "jdbc:oracle:thin:@host:1521:dbname";
        Connection v_connection;
        String updateString = "UPDATE Dozent SET ort = ?";
        PreparedStatement updateOrt;
        try {
            Class.forName("oracle.jdbc.driver.OracleDriver");
        } catch(java.lang.ClassNotFoundException e) {
            System.err.print("ClassNotFoundException: ");
            System.err.println(e.getMessage());
        }
        try {
            v_connection =
                DriverManager.getConnection(url, "benutzer", "passwort");
            updateOrt = v_connection.prepareStatement(updateString);
            updateOrt.setString(1, "Berlin");
            updateOrt.executeUpdate();
            System.out.println("Update ist erfolgt");
            // Statement und Verbindung schließen
            updateOrt.close();
            v_connection.close();
        } catch(SQLException ex) {
            System.err.println("SQLException: " + ex.getMessage());
        }
    }
}

```

Eigenschaften des CallableStatement-Objekts Das CallableStatement-Objekt dient zum Aufruf von Stored Procedures aus der Datenbank. Diese Prozeduren können in Java selbst (vgl. Abschnitt 6.2.2, SQLJ) oder in einer Erweiterung von SQL wie PL/SQL von Oracle (vgl. Abschnitt 7.1) und MySQL (vgl. Abschnitt 7.2) beschrieben sein. Das CallableStatement-Objekt erbt vom PreparedStatement-Objekt. Man unterscheidet bei den Parametern folgende Typen:

- IN-Parameter: nehmen Werte auf, die weiterverarbeitet werden
- OUT-Parameter: enthalten Rückgabewerte
- IN/OUT-Parameter: nehmen Übertragungsparameter auf, deren Wert verändert und an das aufrufende Programm zurückgegeben werden kann

Ein vollständiges Beispiel für die Verwendung des CallableStatement:

Beispiel

```

import java.sql.*;
public class beispiel_callableState {
public static void main(String[] args) {
    String url = "jdbc:oracle:thin:@host:1521:dbname";
    String aufruf = "{call setze_zeitstempel(?)}";
    CallableStatement aufruf_proc = null;
    try {
        Class.forName("oracle.jdbc.driver.OracleDriver");
    } catch(java.lang.ClassNotFoundException e) {
        System.err.print("ClassNotFoundException: ");
        System.err.println(e.getMessage());
    }
    try {
        Connection con =
        DriverManager.getConnection(url, "sport", "sport");
        CallableStatement rufe_proc = con.prepareCall(aufruf);
        rufe_proc.setInt(1,8);
        rufe_proc.executeQuery();

        System.out.println("Procedure wurde schon aufgerufen!");
        // Statement und Verbindung schließen
        con.close();
    } catch(SQLException ex) {
        System.err.println("SQLException: " + ex.getMessage());
    }
}
---Verwendete PL/SQL-Prozedur
CREATE OR REPLACE PROCEDURE setze_zeitstempel(Tage in INTEGER)
IS
BEGIN
    UPDATE Lagerbestand SET Zeitstempel = SYSDATE + Tage;
END;
/

```

6.2.1.6 Verarbeitung der Ergebnismenge im ResultSet

Bei den Statement-Objekten oder PreparedStatement-Objekten entstehen mit dem ResultSet Ergebnismengen, die weiterverarbeitet werden müssen. Ein ResultSet stellt verschiedene Methoden zur Verfügung, in der Tabelle zu navigieren. Vom Grundsatz her entspricht das Konzept der ResultSets dem der CURSOR, welches in PL/SQL im Abschnitt 7.1.4 ausführlich erläutert wird. Es werden hier jedoch wesentlich mehr Methoden angeboten. Die wichtigsten finden Sie in der folgenden Tabelle.

Tabelle 6.1

Die wichtigsten Methoden des ResultSet

Methode	Beschreibung
next()	Stellt den Zeiger zu Beginn auf die erste Zeile und rückt danach jeweils um eine Zeile vor. Diese Methode hat den booleschen Rückgabewert TRUE, wenn sich noch Zeilen im Tabellen-Array befinden, und FALSE, wenn die letzte Zeile erreicht ist.
first()	Stellt den Zeiger auf die erste Zeile.
last()	Stellt den Zeiger auf die letzte Zeile.
previous()	Stellt den Zeiger eine Position zurück.
absolute(n)	Stellt den Zeiger auf die n-te Position.

Schon in unserem ersten JDBC-Beispiel haben wir ein ResultSet-Objekt verwendet:

```

while (v_resultset.next()) {
    String s1 = v_resultset.getString(1);
    System.out.println(s1 );
}

```

Neben der Methode „next()“ wird die Methode „getString()“ benutzt, die den ersten Wert der Zeilen des ResultSet in den String s1 abspeichert. Die SQL-Datentypen werden also auf Java-Datentypen abgebildet. Leider sind die Datentypen bei den verschiedenen Datenbanken nicht einheitlich, so dass es an dieser Stelle beim Wechsel der Datenbank zu Problemen kommen kann.

Tabelle 6.2

Einige Datenkonvertierungen von SQL nach Java über die getXXX-Methoden

	INTEGER	DOUBLE	BIT	CHAR	VARCHAR	BINARY	DATE	CLOB	BLOB
getByte	X	X	X	X					
getInt	X	X	X	X	X				
getLong	X	X	X	X	X				
getFloat	X	X	X	X	X				
getDouble	X	X	X	X	X				
getBoolean	X	X	X	X	X				
getString	X	X	X	X	X				
getDate							X		

	INTEGER	DOUBLE	BIT	CHAR	VARCHAR	BINARY	DATE	CLOB	BLOB
getBytes						X			
getObject	X	X	X	X	X				
getByte	X	X	X	X	X	X	X		
getBlob							X		
getBlob								X	

Die fett formatierten Felder sind zu bevorzugen. Neben den hier beschriebenen Methoden bietet das ResultSet noch eine Vielzahl von Methoden und Eigenschaften, die über den Umfang dieses Buchs hinausführen.³⁶

6.2.1.7 Transaktionsverwaltung

Da Datenbanksysteme für den Multiuser-Einsatz konzipiert sind und ein hohes Maß an Datensicherheit bieten, ist der Begriff der Transaktion grundlegend (vgl. Kapitel 8). Dieses Prinzip muss natürlich auch in einer Java-DB-Verbindung beibehalten werden. In SQL gibt es zu diesem Zweck die Anweisungen COMMIT (schreibt die Änderungen fest) und ROLLBACK (rollt die Änderungen auf einen konsistenten Zustand zurück). Nach einem Verbindungsauflauf ist in JDBC erst einmal der Autocommit-Modus auf TRUE gesetzt. In diesem Modus wird jede SQL-Anweisung als Transaktion behandelt, das heißt einzeln in die Datenbank geschrieben. Wenn man mehrere Datenzugriffe zu einer Transaktion bündeln möchte, setzt man den AUTOCOMMIT-Modus auf FALSE. Dies geschieht mit der Methode setAutoCommit des connection-Objekts:

```
connection.setAutoCommit(false);
```

Neben dieser Methode hat das connection-Objekt auch die Methoden „commit“ und „rollback“, die den SQL-Anweisungen COMMIT und ROLLBACK entsprechen:

```
connection.commit();
connection.rollback();
```

In SQL ist auch das Setzen von Sperren über den Befehl SET TRANSACTION ISOLATION LEVEL vorgesehen (vgl. Abschnitt 8.3). Das Connection-Objekt stellt Methoden zur Verfügung, über die man den Isolationsgrad abfragen und setzen kann.

³⁶ Eine vollständige Übersicht über alle JDBC-Klassen findet man in [White et al. 2000].

Isolationsgrade in JDBC und SQL		
Isolationsgrad	Wert	Beschreibung
TRANSACTION_NONE	0	Es werden keine Sperren in der DB gesetzt.
TRANSACTION_READ_UNCOMMITTED	1	Lesende Transaktionen verursachen keine Sperren.
TRANSACTION_READ_COMMITTED	2	Lesende Transaktionen verursachen Sperren.
TRANSACTION_SERIALIZABLE	3	Transaktionen werden geblockt und hintereinander ausgeführt.

Wenn das Datenbanksystem und der JDBC-Treiber diese Funktion unterstützen, bietet das Connection-Objekt die Möglichkeit, den Isolationsgrad zu lesen und zu verändern:

```
int isoGrad = verbindung.getTransactionIsolation();
connection.setTransactionIsolation(TRANSACTION_NONE);
```

6.2.1.8 Metadaten eines DBMS

Die Informationen über die Datenbank, welche Benutzer es gibt, welche Tabellen angelegt sind, die Datenbankversion etc., sind in relationalen Datenbanksystemen in der Datenbank selbst gespeichert. Diese Informationen können über die Klasse DatabaseMetaData aus der Datenbank abgerufen werden. Insgesamt enthält diese Klasse seit JDBC 2.0 ca. 150 verschiedene Methoden, um Informationen abzurufen. Instantiiert wird ein Objekt der Klasse DatabaseMetaData über die Methode getMetaData des connection-Objekts:

```
DatabaseMetaData dbmd = verbindung.getMetaData();
```

Die verschiedenen Methoden von DatabaseMetaData liefern dann die gewünschten Informationen, z.B. die URL der Datenbank:

```
String URL = dbmd.getURL;
```

Es können auf diese Weise z.B. folgende Informationen abgerufen werden:

- Welche JDBC-Treiber-Version wird benutzt?
- Welche Tabellen sind angelegt?
- Welche Rechte gibt es auf den Tabellen der Datenbank?
- Maximale Anzahl der Spalten in einer Tabelle
- Maximale Länge eines SQL-Statement
- Ist die Datenbank im Nur-Lese-Modus?
- Liste aller mathematischen Funktionen des DBMS
- Liste aller Zeit- und Datumsfunktionen
- Mit welchem User ist die Anwendung eingeloggt?

Ein Objekt der Klasse `ResultSetMetaData` ist ähnlich aufgebaut wie `DatabaseMetaData`, mit dem Unterschied, dass die Informationen aus einer `SELECT`-Abfrage abgeleitet werden.

Das folgende Beispiel liefert unterschiedliche Informationen zur Tabelle Kunden aus der Byce & Co.-Datenbank:

Beispiel

```
import java.sql.*;
public class RSMetaDataMethods {
    public static void main(String args[]) {
        String url = "jdbc:oracle:thin:@host:1521:dbname";
        Connection con;
        Statement stmt;
        try {
            Class.forName("oracle.jdbc.driver.OracleDriver");
        } catch(java.lang.ClassNotFoundException e) {
            System.err.print("ClassNotFoundException: ");
            System.err.println(e.getMessage());
        }
        try {
            con = DriverManager.getConnection(url,
                "fahrrad", "fahrrad");
            stmt = con.createStatement();
            ResultSet rs = stmt.executeQuery
                ("SELECT * FROM Kunden");
            ResultSetMetaData rsmd = rs.getMetaData();
            int numberOfColumns = rsmd.getColumnCount();
            for (int i = 1; i <= numberOfColumns; i++) {
                String colName = rsmd.getColumnName(i);
                String tableName = rsmd.getTableName(i);
                String name = rsmd.getColumnTypeName(i);
                boolean caseSen = rsmd.isCaseSensitive(i);
                boolean writable = rsmd.isWritable(i);
                System.out.println("Information for column " + colName);
                System.out.println("Column is in table " + tableName);
                System.out.println("DBMS name for type is " + name);
                System.out.println("Is case sensitive: " + caseSen);
                System.out.println("Is possibly writable: " + writable);
                System.out.println("");
            }
            while (rs.next()) {
                for (int i = 1; i <= numberOfColumns; i++) {
                    String s = rs.getString(i);
                    System.out.print(s + " ");
                }
                System.out.println("");
            }
            stmt.close();
            con.close();
        } catch(SQLException ex) {
    }
```

```
System.err.println ("SQLException: " + ex.getMessage());
```

6.2.1.9 Ausnahmebehandlung in JDBC

Da in JDBC erst zur Laufzeit eine Übersetzung der SQL-Anweisungen erfolgt, ist mit Laufzeitfehlern zu rechnen, die eine SQLException verursachen. Die Ausführung eines SQL-Befehls muss daher immer in einem try/catch-Block erfolgen. Der catch-Block enthält die Fehlerbehandlung und das Auffangen einer SQLException.

Eine SQLException besteht aus

- einer Beschreibung des Fehlers, die durch die Methode getMessage () geliefert wird,
 - einem SQL-Status, der durch die Methode getStatus () geliefert wird sowie
 - einer herstellerabhangigen Fehlernummer, die durch die Methode getErrorCode () geliefert wird.

Bei einer SELECT-Anweisung auf eine nicht vorhandene Tabelle bekommt man z.B. folgende Werte:

- `getStatus()` liefert den Wert 42000.
 - `getMessage()` liefert „Tabelle oder View nicht vorhanden“.
 - `getErrorCode` liefert die Oracle-Fehlernummer 00942

Neben den SQLExceptions, die durch Datenbankfehler ausgelöst werden, gibt es noch die Unterklassen SQLWarning (von SQLException) und DataTruncation (von SQLException). SQLWarnings führen nicht zu Unterbrechungen des Programms. DataTruncation enthalten Informationen über Read/Write-Informationen. Beide Typen werden in dieser Kurzeinführung nicht behandelt. Der interessierte Leser sei wieder auf [White et al. 2000] verwiesen.

6.2.2 SQL

1997 gründete sich ein SQL-Konsortium³⁷ verschiedener Firmen und Einrichtungen der IT-Branche, mit dem Ziel, die Zusammenarbeit von Java und relationalen Datenbanken zu verbessern. Zu den beteiligten Firmen gehörten Compaq (Tandem), IBM, Informix, Micro Focus, Microsoft, Oracle, Sun und Sybase. Die Idee war, die Einbettung von SQL in eine Programmiersprache, die zum Beispiel mit Embedded C schon bestand, auch auf Java auszudehnen. Während bei JDBC dynamisches SQL eingesetzt wird, d.h., die SQL-Anweisungen werden als Zeichenkette zur Datenbank gesendet und zur Laufzeit übersetzt, handelt es sich bei SQLJ um statisches SQL. Der Nachteil bei dynamischem SQL ist, dass Fehler eben erst zur Laufzeit erkannt werden. Bei statischem SQL werden die SQL-Anweisungen vorab übersetzt (in der SQLJ-Sprechweise macht das der „Translator“) und in Java-Code überführt, der dann vom normalen Java-Übersetzer weiterverarbeitet wird. SQLJ wurde in drei Teile gegliedert:

37 Der Webserver des SQLJ-Konsortiums ist unter der URL <http://www.sqlj.org/> zu finden.

- Part 0: Embedded SQL: Einbindung von statischen SQL-Statements in ein Java-Programm

- Part 1: Java Stored Routines: Nutzung von statischen Java-Methoden als SQL Stored Procedures

- Part 2: Java Data Types: Verwendung von Java-Klassen als SQL Abstract Data Types

6.2.2.1 SQLJ-Klauseln und HOST-Variablen (Part 0)

SQLJ-Klauseln und HOST-Variablen sind als Bestandteil von Part 0 von SQLJ grundlegend für die anderen Bereiche. Es handelt sich bei den SQLJ-Klauseln um statische SQL-Anweisungen, die in Java eingebettet sind. Diese statischen SQLJ-Anweisungen werden folgendermaßen verarbeitet:

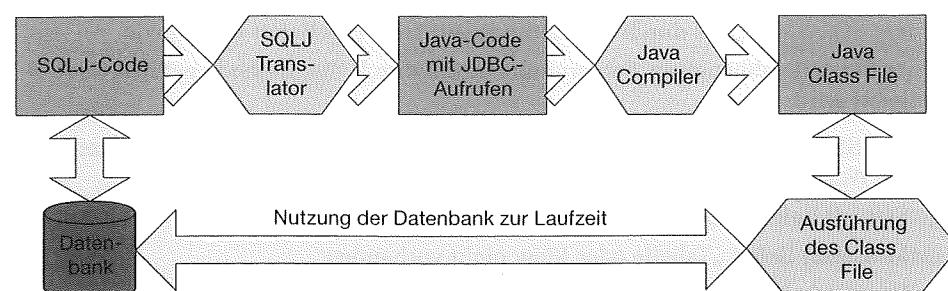


Abbildung 6.11: SQLJ-Lebenszyklus

6.2.2.2 Ablauf der Verarbeitung von SQLJ-Dateien

Der SQLJ-Translator überprüft schon zur Compilezeit, ob die SQL-Syntax korrekt ist, die Datentypen übereinstimmen und die Tabellendefinitionen der Schemadefinition entsprechen. Der Translator wird über die Kommandozeilenprozedur „sqlj“ aufgerufen und auf eine Datei mit der Endung *.sqlj , z.B. test.sqlj, angewandt. Ergebnis ist eine *.java-Datei mit JDBC-Aufrufen. Diese Datei, z.B. test.java, wird wie jede andere Java-Datei mit dem Java-Compiler (z.B. javac test.java) in eine Class-Datei übersetzt, die mittels der Laufzeitumgebung „java“ gestartet werden kann.

6.2.2.3 SQLJ-Klauseln

Mit SQLJ-Klauseln wird SQL-Code in Java eingebettet und in einer Datei mit der Endung *.sqlj abgespeichert. Man unterscheidet Deklarationen von Iteratoren und Kontexten (vgl. Abschnitt 6.2.2) und ausführbare SQL-Anweisungen, die in diesem Kapitel behandelt werden. SQLJ-Klauseln haben die Gestalt #sql{...}; und können mehrere Zeilen, allerdings nur eine SQL-Anweisung, umfassen. Sie werden mit einem Semikolon abgeschlossen. In den geschweiften Klammern sind SQL-Anweisungen ohne Semikolon enthalten.

Beispiel

```
#sql { DELETE FROM Beispieldatenebene ; }
#sql { CREATE TABLE Test (Zahl NUMBER) ; }
```

SQLJ-Klauseln können SELECT-Anweisungen, andere DML-Anweisungen (wie INSERT, UPDATE und DELETE), DDL-Anweisungen (wie CREATE TABLE) und den Aufruf von gespeicherten Routinen (vgl. Kapitel 7) enthalten. Auch Transaktionsverarbeitungen mittels ROLLBACK und COMMIT sowie SET-Anweisungen sind vorgesehen.

```
#sql { COMMIT ; }
#sql { SET TRANSACTION READ ONLY ; }
#sql { SET TRANSACTION ISOLATION LEVEL 1 ; }
```

Um z.B. SELECT-Anweisungen zu verarbeiten, benötigt man ein Verfahren, um das Ergebnis der SELECT-Anweisungen weiterzuverarbeiten. Diese Möglichkeit eröffnen die HOST-Variablen.

HOST-Variablen

HOST-Variablen sind Java-Variablen, die den Austausch zwischen der HOST-Sprache (hier Java) und der SQL-Anweisung ermöglichen. Sie werden durch einen Doppelpunkt (:Variablename) gekennzeichnet.

Beispiel

```
String meine_variable;
#sql {DROP TABLE Test;};
#sql {CREATE TABLE Test (was_auch_immer VARCHAR2(100))};
#sql {INSERT INTO Test (was_auch_immer) VALUES
      ('hallo_')||TO_CHAR(SYSDATE));
#sql {SELECT was_auch_immer INTO :meine_variable FROM Test};
```

Dieses Beispiel enthält mehrere SQL-Anweisungen, deren Ergebnis als SELECT-Anweisung über die HOST-Variablen „:meine_variable“ ausgegeben wird.

Die Richtung der Übertragung von Daten kann mit „IN“, „OUT“ und „IN OUT“ ähnlich wie in PL/SQL angegeben werden, wobei standardmäßig „OUT“ bei „SELECT INTO :Variable“ und „IN“ bei allen anderen SQL-Anweisungen angenommen wird. Die Richtungsangabe steht durch Leerzeichen trennt vor der Variablen.

:IN variable

Eine SQLJ-Klausel kann auch Werte zurückliefern, wenn z.B. eine SQL-Anweisung einen Wert ergibt:

```
#sql Ergebnis = {SQL-Anweisung mit Rückgabewert}
#sql Ergebnis = {SELECT COUNT(*) FROM Angestellte}
```

In diesem Fall wird die Zielvariable „Ergebnis“ außerhalb der geschweiften Klammern ohne Doppelpunkt angegeben.

Ähnlich funktioniert auch der Aufruf einer in der Datenbank gespeicherten Funktion:

```
#sql Ergebnis = | VALUES (Funktion(Parameterliste)) |
```

Mit CALL werden in der Datenbank gespeicherte Prozeduren aufgerufen:

```
#sql | CALL Prozedur(Parameterliste)|
```

Für weitere Einzelheiten bezüglich SQLJ verweisen wir auf [Saake et al. 2000].

6.2.2.4 Iteratoren (Part 0)

Ein Iterator ist ein Mechanismus, der ähnlich wie ein CURSOR in PL/SQL (vgl. Abschnitt 7.1.4) oder ein ResultSet in JDBC (vgl. Abschnitt 6.2.1) arbeitet. Er regelt den Zugriff auf die Ergebnismenge einer SELECT-Anweisung, falls nicht nur ein Tupel, sondern mehrere erzeugt werden.

Es gibt zwei Typen von Iteratoren: benannte Iteratoren und Positionsiteratoren. Bei benannten Iteratoren ist eine durch Komma getrennte Liste von Attributnamen, bestehend aus Typ und Bezeichner, anzugeben, die den Spalten der Ergebnisrelation entsprechen. Die Reihenfolge der Iteratorattribute spielt keine Rolle, da jedem Attribut eindeutig eine Spalte der Ergebnisrelation zugeordnet ist. Diese Zuordnung ist nicht case-sensitiv.

Der Ablauf sieht bei benannten Iteratoren folgendermaßen aus:

1. Schritt: Deklaration des Iterators

```
#sql iterator PersonName(String Vorname, String Nachname);
```

2. Schritt: Vereinbarung eines Iterator-Objekts

```
PersonName iter;
```

3. Schritt: Ausführung der SELECT-Anweisung

```
#sql iter=(SELECT Vorname, Nachname FROM Person);
```

4. Schritt: Navigation über die Ergebnismenge und Auslesen der Daten

```
while(iter.next()){
    System.out.println("Vorname = " + iter.Vorname());
    System.out.println("Nachname = " + iter.Nachname()); }
```

5. Schritt: Freigabe der Ressourcen durch Schließen des Iterators

```
iter.close();
```

Positionsiteratoren werden nicht durch einen Namen, sondern durch ihre Position bestimmt, d.h., es wird nur eine Liste der verwendeten Typen deklariert. Der Ablauf ändert sich hier geringfügig, nämlich bei der Deklaration des Iterators und beim Auslesen der Daten.

1. Schritt: Deklaration des Iterators

```
#sql iterator PersonName(String , String );
```

Der 2. Schritt, der 3. Schritt und der 5. Schritt sind analog zum benannten Iterator, nur bei Schritt 4 ergibt sich eine leicht veränderte Syntax:

4. Schritt: Navigation über die Ergebnismenge und Auslesen der Daten bei Positionsiteratoren

```
while
    #sql {FETCH :iter INTO :Vorname, :Nachname};
    if (:iter.endFetch()) break;
    System.out.println("Vorname " + :Vorname);
    System.out.println("Nachname " + :Nachname);
```

6.2.2.5 Kontexte (Part 0)

Bisher wurde noch nicht behandelt, wie eine Datenbankverbindung innerhalb SQLJ gehandhabt wird. Eine Verbindung wird in SQLJ grundsätzlich durch einen Verbindungskontext, eine Instanz der Klasse sqlj.runtime als Connection-Context, repräsentiert. Der Verbindungskontext spezifiziert die Datenbank mit den assoziierten Schemata und den Verbindungsinformationen.

```
import sqlj.runtime.*;
import sqlj.runtime.ref.*;
import java.sql.*;
public class Sqlj3 {
    static String driverClass = "oracle.jdbc.driver.OracleDriver";
    static String url = "jdbc:oracle:thin:@localhost:1521:orcl1";
    public static void main (String [] args) {
        try {
            Class.forName(driverClass);
        }catch (ClassNotFoundException exc) {
            System.out.println(exc.getMessage());
            System.exit(1);
        }
        try {
            Connection con = DriverManager.getConnection
                (url, "user", "password");
            DefaultContext ctx = new DefaultContext (con);
            DefaultContext.setDefaultContext(ctx);
            #sql iterator PersonName(String Vorname, String Nachname);
            PersonName iter;
            #sql iter=(SELECT Vorname,Nachname FROM Angestellte);
            while(iter.next()){
                System.out.println("Vorname =" + iter.Vorname());
                System.out.println("Nachname=" + iter.Nachname());
            }
            iter.close();
        } catch (SQLException exc ) {
            System.out.println (exc.getMessage());
        }
    }
}
```

Dieses Beispiel ist der vollständige Sourcecode einer kleinen SQLJ-Implementierung. Zuerst werden die benötigen Bibliotheken sqlj.runtime und java.sql importiert. Der JDBC-Treiber wird wie in einer JDBC-Verbindung in ein Connection-Objekt „con“ geladen. Die eigentliche Datenbankverbindung geschieht über

```
DefaultContext ctx = new DefaultContext (con);
```

Mit

```
DefaultContext.setDefaultContext(ctx);
```

wird der Kontext zum default erklärt. Über

```
#sql [ctx] | SQL Operation |;
```

ist es möglich, einer anderen SQL-Anweisung diesen Kontext explizit zuzuordnen. Ein Kontext kann auch mehrfach genutzt werden, um die Performance zu verbessern, da der Aufbau von Datenbankverbindungen zeitaufwändig ist. Über verschiedene Kontexte können in einem SQLJ-Programm auch verschiedene Datenbankverbindungen, die einen anderen Kontext besitzen, aufgebaut werden.

Eigenschaften von Kontexten:

- Mehrere Verbindungskontexte können gleichzeitig instanziert und benutzt werden.
- Verschiedene Verbindungskontext-Klassen können zur Partitionierung von Anweisungen, die in verschiedenen Schemata ausgeführt werden, genutzt werden.

Auf diese Einzelheiten wird im Rahmen dieser Einführung nicht weiter eingegangen. Der interessierte Leser findet eine ausführlichere Beschreibung von SQLJ z.B. in [Saake et al. 2000] oder [Melton 2000].

6.2.2.6 Gespeicherte Funktionen und Prozeduren (Part 1)

In SQLJ Part 1 wird gefordert, anstelle von PSM-Stored-Routinen Java Stored Procedures (vom Typ SQLJ oder auch andere wie JDBC) direkt in die Datenbank einzuspielen und wie andere gespeicherte Prozeduren aufzurufen.

Java Stored Procedures haben einige Restriktionen:

- Es ist keine direkte Interaktion mit dem Benutzer möglich, da die Prozeduren ja in der Datenbank gespeichert sind und ausgeführt werden.
- Es besteht immer eine DEFAULT-Verbindung zur Datenbank, die nicht veränderbar ist.
- Prozeduren und Methoden werden als Klassenmethoden implementiert.

Die Benutzung von Java Stored Procedures unter Oracle erfordert mehrere Schritte (Implementierung, Übersetzung, Installation im Server mittels loadjava und Registrierung) und wird hier nicht weiter vertieft, der interessierte Leser findet Einzelheiten in der Oracle-Dokumentation³⁸ oder bei [Saake 2000].

6.2.2.7 Java-Klassen und SQL-Datentypen (Part 2)

Bei JDBC, SQLJ und beim Einsatz von Java Stored Procedures und Functions tritt das Problem auf, welche Datentypen von SQL und Java, auch Java-Klassen, aufeinander abgebildet werden. Die DEFAULT-Abbildungen sind in der folgenden Tabelle dargestellt.

Tabelle 6.4

Abbildung von SQL-Datentypen auf Java-Datentypen

SQL Datatypes	Standard Java Types	SQL Datatypes	Standard Java Types
CHAR	java.lang.String	NUMBER	int
VARCHAR2	java.lang.String	NUMBER	long
LONG	java.lang.String	NUMBER	float
NUMBER	java.math.BigDecimal	NUMBER	double
NUMBER	java.math.BigDecimal	RAW	byte[]
NUMBER	boolean	LONGRAW	byte[]
NUMBER	byte	DATE	java.sql.Date
NUMBER	short	DATE	java.sql.Time
BLOB	java.sql.Blob	DATE	java.sql.Timestamp
CLOB	java.sql.Clob	user-defined object	java.sql.Struct
user-defined reference	java.sql.Ref	user-defined reference	java.sql.Ref
user-defined collection	java.sql.Array		

6.2.2.8 Vergleich von JDBC und SQLJ

In diesem Kapitel werden kurz die Eigenschaften von JDBC und SQLJ verglichen und der Sourcecode gegenübergestellt.

Tabelle 6.5

Vergleich Eigenschaften von JDBC und SQLJ

Eigenschaften von JDBC	Eigenschaften von SQLJ
Dynamisches SQL	Statisches SQL
Hohe Flexibilität	Geringe Flexibilität
Fehlererkennung zur Laufzeit	Fehlererkennung zur Compile-Zeit

³⁸ z.B. http://download-east.oracle.com/docs/cd/B14117_01/java.101/b12021/toc.htm, 29.12.2006

Tabelle 6.6

Vergleich Quellcode von JDBC und SQLJ**JDBC-Quellcode**

```
java.sql.CallableStatement stmt;
Connection conn;
ResultSet results;
conn = DriverManager.getConnection
    ("jdbc:default");
stmt = conn.prepareStatement
    ("SELECT ename FROM emp
     WHERE sal > ? AND deptno = ?");
stmt.setInt(1, salparam);
stmt.setInt(2, deptnparam);
results = stmt.executeQuery();
```

SQLJ-Quellcode

```
ResultSet results;
#sql results =
  {SELECT ename FROM emp
   WHERE sal > :salparam
   AND deptno =:deptnparam};
```

ZUSAMMENFASSUNG

Dieses Kapitel hat objektrelationale Erweiterungen, allerdings aus ganz unterschiedlichen Bereichen zum Inhalt, zum einen die objektrelationalen Erweiterungen der SQL-Syntax, zum anderen die Anbindung von SQL an die objektorientierte Programmiersprache Java mittels JDBC und SQLJ und außerdem die objektrelationale Abbildung. Der Abschnitt 6.1. beschäftigte sich mit den wichtigsten Aspekten von objektrelationaler SQL, nämlich den benutzerdefinierten Datentypen, typisierten Tabellen, Tupeltabellen, Vererbung, Anfragen und den Grenzen der relationalen Modellierung. In der Praxis ist es vielfach üblich, ein objektrelationales konzeptionelles Schema zu bilden, da sich mit diesem Verfahren die Realität besser abbilden lässt. Allerdings wird dann doch kein objektrelationales Datenbankschema abgeleitet, sondern statt dessen ein relationales Datenbankschema. In diesem Fall hilft dann eine objektrelationale Abbildung weiter, die in Abschnitt 6.1.11. vorgestellt wurde. In den gleichen Themenkreis gehört auch die objektrelationale Anwendungsprogrammierung, da in der Regel objektorientierte Sprachen wie Java benutzt werden. Wir haben den Grundaufbau einer JDBC-Verbindung, Metadaten, Transaktionshandling und Fehlerbehandlung sowie SQLJ als alternative Java-Variante in Abschnitt 6.2 dargestellt.

Weiterführende Literatur

Leider reicht der Raum für eine gründliche Vertiefung nicht aus. Wer daran interessiert ist, findet herstellerunabhängige Literatur über objektrelationale Datenbanken in [Türker 2003], [Türker et al. 2006] und [Stonebraker et al. 1999], worauf im Text schon mehrfach verwiesen wurde. Objektrelationale Anwendungsprogrammierung mittels JDBC-Anbindung und SQLJ gehören zum SQL2003 Standard und sind daher auch z.B. in [Türker 2003] oder sehr ausführlich in [Saake 2000] oder [White et al. 2000] beschrieben. Die Oracle-Implementierungen sind in den beiden Büchern von Hohenstein³⁹ oder in der Oracle-Dokumentation⁴⁰ selber ausführlich behandelt. Für MySQL finden Sie weitergehende Informationen in der MySQL-Dokumentation.

³⁹ [Hohenstein 2000] und [Hohenstein et al. 2003]

⁴⁰ vgl. <http://www.oracle.com>, 29.12.2006



Übungsaufgaben



Aufgaben zum Abschnitt 6.1 (Objektrelationale Datenbanken):

Die Aufgaben zu Abschnitt 6.1 sind nur mit einer Oracle-Datenbank lösbar, da MySQL noch keine objektrelationalen Erweiterungen besitzt.

- 1** a. Erweitern Sie die Tabelle „Spieler“ aus Ihrer Rollo-Datenbank aus Aufgabe 5.1 um eine Spalte „Bilder“ vom Datentyp BLOB und eine Spalte „Lebenslauf“ vom Typ XMLType.
b. Fügen Sie ein Bild und einen Lebenslauf für einen Spieler ein!
c. Was passiert, wenn das XML-Dokument nicht gültig ist?
d. Suchen Sie mit einem xpath-Ausdruck und extractNode in der Tabelle „Spieler“ nach einem bestimmten Ausdruck.
- 2** Erstellen Sie zum Datenbankschema Rollo aus Aufgabe.1 in Kapitel 5 drei Typen Personen_t, Spieler_t und Kunden_t, wobei Kunden_t und Spieler_t von Personen_t erben und den selbstdefinierten Typen Adresse_t verwenden.
- 3** Erstellen Sie zu Ihrem Rollo-Datenbankschema drei Typen für Tore, Spiele und Nationen. Für Spiele wird ein VARRAY der Länge zwei (Mannschaft1, Mannschaft2) benötigt und die Tore sollen mit den Spielern und den Spielen in einer REF-Beziehung verbunden sein.
 - a. Erstellen Sie einen Typ Spiele_t. Für Spiele wird ein VARRAY der Länge zwei (Mannschaft1, Mannschaft2) benötigt.
 - b. Erstellen Sie zu den Typen aus Aufgabe 6.2 a) und 6.3. b) typisierte Tabellen und eine Tupel-Tabelle Tore. Die Tore sollen mit Spieler und Spiele mit einer REF-Beziehung verbunden sein.
- 4** a. Erstellen Sie zu den Typen aus Aufgabe 2 und 3 typisierte Tabellen und fügen Sie in jede Tabelle mindestens einen Datensatz ein, unter anderem mindestens einen Datensatz für ein Spiel, an dem Deutschland teilnimmt. Die Tabelle „Spiele“ soll die Spaltenbedingung erfüllen, dass die Anzahl_Zuschauer größer als Null ist. Alle Tabellen haben vom System generierte OIDS.
b. Fügen Sie die Skripten aus den Aufgaben 2 bis 4 in ein SQL-Skript ein, das sich mehrfach aufrufen lässt. Dazu müssen DROP-Anweisungen in der korrekten Reihenfolge ergänzt werden.
- 5** Schreiben Sie eine SELECT-Anweisung, die aus der Tabelle „Spiele“ alle Spiele selektiert, an denen Deutschland teilnimmt.

- 6** Erzeugen Sie Typen für Bestellungen und Karten, die Karten als geschachtelte Tabelle von Bestellungen vorsieht, sowie die zugehörigen Tabellen (STORE AS NESTED TABLE). Fügen Sie einige Datensätze in die Tabellen ein.

- 7** Betrachten Sie ► Abbildung 3.39 aus Kapitel 3, die ein Buch beschreibt.
 - a. Mit welchen objektrelationalen Typkonstruktoren lässt sich dieses Beispiel darstellen?
 - b. Wie kann man dieses Beispiel unter Oracle-SQL abbilden?

Aufgaben zum Abschnitt 6.2 (JDBC und SQLJ):

- 8** Erstellen Sie eine JDBC-Prozedur, die alle Daten aus der Tabelle „Spiele“ ausliest und in einem Systemfenster ausgibt.
- 9** Erstellen Sie eine JDBC-Prozedur, die die Tabellen, die mit Ihrer Datenbankkennung angelegt sind, mit den zugehörigen Spaltennamen ausgibt!
- 10** Schreiben Sie ein Datenbank-Abfragetool als Java-Swing-Applikation, das eine URL der Form „jdbc:oracle:thin:benutzer/passwort@domain:port:sid“ sowie eine SELECT-Abfrage (ohne abschließendes Semikolon) entgegennimmt. Durch das Anwählen eines Buttons „Abschicken“ wird die im SELECT-String enthaltene Abfrage an die durch die URL verbundene Datenbank geschickt und in einem JTable, der auf einem AbstractTableModel aufbaut, angezeigt. Bei Fehlern wird mindestens eine Ausgabe des Fehlertextes im Systemfenster, besser in der Applikation selbst, erzeugt.

Bei Änderungen der Abfrage bzw. der URL soll nur dann eine neue Verbindung aufgebaut werden, falls sich die URL verändert hat. Bei einer veränderten Abfrage wird nur die JTable-Ausgabe aktualisiert.

Weitere Kontrollfragen zu diesem Kapitel finden Sie unter der Companion-Webseite des Pearson-Verlages <http://www.pearson-studium.de/> auf der Begleitseite unseres Buches. Wählen Sie dort bitte im Multiple-Choice-Test das Fach „DBS“ und den Punkt „Kapitel6/Objektrelationale Erweiterungen von SQL2003“ aus. Hier finden Sie auch einen interaktiven JDBC-Trainer.

