Teil V

Kombination von SQL mit anderen Programmiersprachen

17 EINBETTUNG VON SQL IN HÖHERE PROGRAMMIERSPRACHEN

Neben der Erweiterung von SQL um prozedurale Konzepte (siehe Abschnitt
Bei der Einbettung von SQL in eine bestehende Programmiersprache – etwa C – werden SQL-Anweisungen durch Schlüsselwörter, z. B. `EXEC SQL`, eingeleitet und prinzipiell wie Anweisungen der Programmiersprache behandelt.
Dabei ergibt sich aufgrund der unterschiedlichen Typsysteme und Paradigmen ein *impedance mismatch* (Missverhältnis): Den Tupelmengen in SQL steht in der Regel kein vergleichbarer Datentyp in der Programmiersprache gegenüber. Des weiteren sind Programmiersprachen in der Regel imperativ, während SQL deklarativ ist. Dieser Konflikt wird in der Praxis dadurch gelöst, dass die Tupel bzw. Attribute auf die Datentypen der Hostsprache abgebildet werden, und Tupelmengen iterativ durch *Cursore* verarbeitet werden.
Bei der Kombination von SQL mit objektorientierten Programmiersprachen ist der konzeptuelle *impedance mismatch* noch größer. Aus dieser Situation entstanden die Konzepte von ODBC (*Open Database Connectivity*) in der Microsoft-Welt, sowie JDBC (*Java Database Connectivity*, vgl. Abschnitt

17.1 Embedded-SQL in C

In diesem Abschnitt wird die Einbettung von SQL-Anweisungen in C beschrieben.[1]
Bei der Einbettung von SQL in C – werden SQL-Anweisungen durch das Schlüsselwort `EXEC SQL` eingeleitet und prinzipiell wie Anweisungen der Programmiersprache behandelt.
Ein solches C-Programm `<name>.pc` mit Embedded-SQL-Statements wird mit Hilfe eines Precompilers in ein C-Programm `<name>.c` transformiert, wobei die `EXEC SQL`-Statements in Aufrufe einer runtime library übersetzt werden. Das C-Programm `<name>.c` wird dann mit dem üblichen C-Compiler weiter übersetzt (siehe Abbildung
Die dabei verwendeten SQL-Statements können entweder fest vorgegeben (statisch), oder im Zuge des Programmablaufs als Strings zusammengesetzt werden (dynamische SQL-Anweisungen). Hier stehen statische SQL-Anweisungen im Vordergrund.

[1] In diesem Abschnitt wird dazu der ORACLE Pro* C/C++ Precompiler Version 2.1 verwendet. Die beschriebene Vorgehensweise zur Einbettung von SQL-Anweisungen lässt sich auch auf andere Programmiersprachen übertragen.

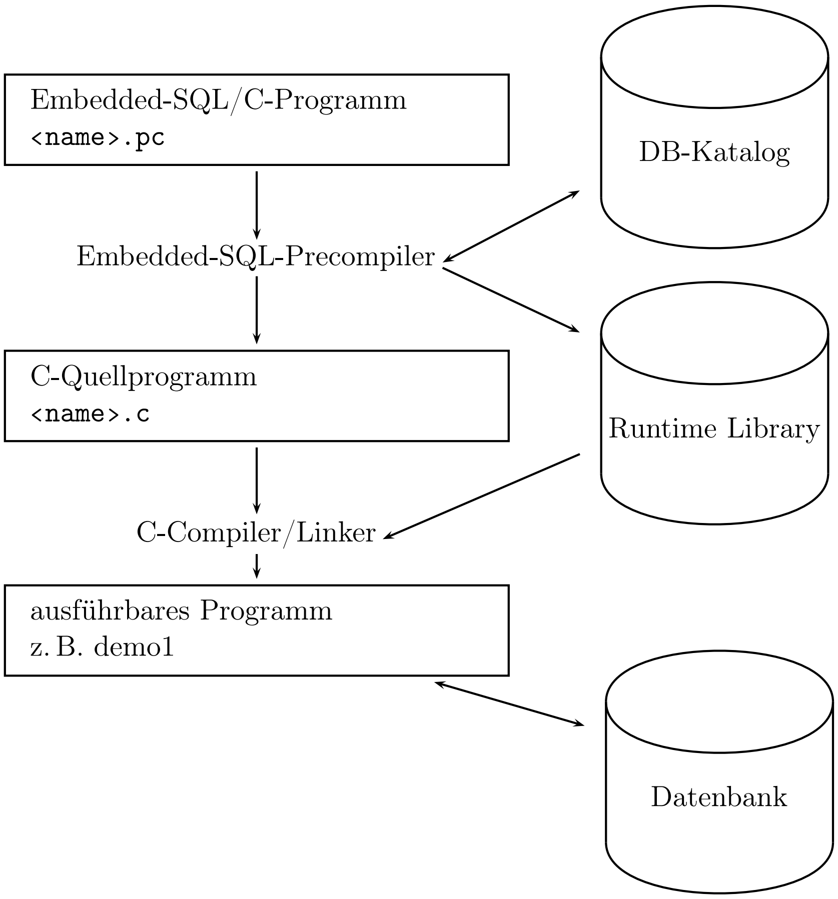

Abbildung 17.1: Entwicklungsschritte von Embedded-SQL Anwendungen

Verbindung zur Datenbank. Im Gegensatz zu SQL*plus und PL/SQL, wo die Verbindung zur Datenbank automatisch gegeben ist, muss in einer Embedded-Anwendung zuerst eine Verbindung zu einer Datenbank hergestellt werden. Die geschieht mit dem `CONNECT`-Statement:

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

wobei `username` und `passwd` Hostvariablen (siehe nächsten Abschnitt) vom Typ `CHAR` bzw. `VARCHAR` sind. Die Verwendung von Hostvariablen für das Einloggen kann *nicht* durch direkte Verwendung von Strings in der `CONNECT` Anweisung umgangen werden, eine Anweisung der Art

```
EXEC SQL CONNECT ‘‘dummy’’ IDENTIFIED BY ‘‘dummypasswd’’; /* nicht erlaubt */
```

schlägt daher fehl.
Alternativ kann das Einloggen auch in der Form

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

erfolgen, wobei `uid` dann die Benutzerkennung, einen slash (“`/`”) und das Passwort enthält, etwa `"dummy / dummypasswd"`. Wenn der Betriebssystemaccount mit dem Datenbankaccount übereinstimmt (wie im Praktikum), kann dabei “`/`” statt dem Benutzernamen verwendet werden.

Hostvariablen. Die Kommunikation zwischen der Datenbank und dem Anwendungsprogramm findet über *Hostvariablen*, d.h. Variablen der umge-

benden Programmiersprache statt. Diese können sowohl in den Anweisungen der Hostsprache als auch in den embedded-SQL-Anweisungen stehen. Innerhalb eines SQL-Statements wird einer Hostvariable immer ein Doppelpunkt (":") vorangestellt (nicht jedoch in der Hostsprache), um sie von Datenbankobjekten abzuheben. Dabei dürfen Hostvariablen in statischen SQL-Anweisungen nicht an Stelle von Datenbankobjekten (Tabellen, Views, etc.) verwendet werden. Hostvariablen, die in der `INTO`-Klausel eines `SELECT`- oder `FETCH`-Statements auftreten, sind *Output*-Variablen (bzgl. des SQL-Statements). Alle anderen Hostvariablen in einem SQL-Statement sind hingegen *Input*-Variablen. Input-Variablen müssen initialisiert werden, bevor sie in einem SQL-Statement verwendet werden.
Hostvariablen können auch in einem `struct` zusammengefasst werden. Die einzelnen Komponenten des `struct` werden dann als Hostvariable betrachtet. Structs dürfen jedoch nicht geschachtelt werden. Bei der Verwendung einer Hostvariable ist darauf zu achten, dass sie zu dem Datentyp des Attributs kompatibel ist, auf das im SQL-Statement Bezug genommen wird. Die Beziehung zwischen C- und ORACLE-Datentypen ist in Abbildung
ORACLE stellt für die Embedded-Programmierung in C einen C-Datentyp `VARCHAR` bereit, der an den internen Datentyp `VARCHAR2` angelehnt ist.

Indikatorvariablen. Jeder Hostvariablen kann eine Indikatorvariable zugeordnet werden, die den aktuellen Status der Hostvariablen anzeigt. Indikatorvariablen sind insbesondere für die Verarbeitung von Nullwerten von Bedeutung (hier äußert sich der Impedance Mismatch, dass herkömmliche Programmiersprachen keine Nullwerte kennen). Sie sind als `short` zu deklarieren; ihre Werte werden wie folgt interpretiert:

Indikatorvariablen für Output-Variablen:

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

Indikatorvariablen für Input-Variablen:

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

Innerhalb eines SQL-Statements (nicht jedoch in der Hostsprache) wird der Indikatorvariable ein Doppelpunkt (":") vorangestellt, dabei muss sie direkt auf die entsprechende Hostvariable folgen. Zur Verdeutlichung kann zwischen den beiden Variablen auch das Schlüsselwort `INDICATOR` stehen (siehe demo1.pc). Wird für Hostvariablen ein `struct` verwendet, so lässt sich diesem auch ein entsprechender `struct` für Indikatorvariablen zuordnen.

Declare-Section. Host- und Indikatorvariablen werdem in der *Declare-Section* deklariert:

C-Datentyp	ORACLE-Datentyp
char char[n] VARCHAR[n] int short long float double	VARCHAR2(n) CHAR(X)
int short long float double char char[n] VARCHAR[n]	NUMBER NUMBER(P,S)
char[n] VARCHAR[n]	DATE
char[n] VARCHAR[n]	LONG
unsigned char VARCHAR[n]	RAW(X)
unsigned char[n] VARCHAR[n]	LONG RAW

Abbildung 17.2: Kompatibilität zwischen C- und ORACLE-Datentypen

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

Deklarationen werden nicht wie SQL-Anweisungen in einen Aufruf der runtime library übersetzt. Je nach Version bzw. Modus des Precompilers können Host- bzw. Indikatorvariablen auch als "gewöhnliche" C-Variablen ohne Declare-Section deklariert werden.

Cursore. Wenn die Anwortmenge zu einer `SELECT`-Anfrage mehrere Tupel enthält, die in einem Anwendungsprogramm verarbeitet werden sollen, geschieht dies über einen Cursor (oder alternativ ein Hostarray). Die Cursoroperationen sind im Prinzip dieselben wie in PL/SQL, sie sind allerdings hier keine "vollwertigen" Befehle, sondern müssen ebenfalls durch `EXEC SQL` eingeleitet werden.
Mit der Deklaration

```
EXEC SQL DECLARE <cursor-name> CURSOR FOR <select-statement>;
```

wird ein Cursor benannt und einer Anfrage zugeordnet.

Für den Umgang mit Cursoren stehen auch hier drei Operationen zur Verfügung:

- `OPEN <cursor-name>;`
  öffnet einen Cursor, indem die entsprechende Anfrage ausgeführt und der Cursor vor das erste Ergebnistupel gesetzt wird.
- `FETCH <cursor-name> INTO <varlist>;`
  setzt den Cursor, solange noch Daten vorhanden sind, auf das nächste Tupel. `FETCH` erzeugt in den folgenden Fällen einen Fehler:
- der Cursor wurde nicht geöffnet bzw. nicht deklariert.
- es wurden keine (weiteren) Daten gefunden.
- der Cursor wurde geschlossen, aber noch nicht wieder geöffnet.
- `CLOSE <cursor-name>;`
  schließt den Cursor.

Für Anfragen, die nur ein einziges Tupel liefern, kann dieses direkt mit einem `SELECT ... INTO :<hostvar>`-Statement einer Hostvariablen zugewiesen werden. In diesem Fall wird impliziert ein Cursor deklariert.

Hostarrays. Mittels *Hostarrays* lässt sich sowohl die Programmierung vereinfachen, als auch der Kommunikationsaufwand zwischen Client und Server reduzieren: Mit einem Hostarray können mit einem einzigen `FETCH`-Zugriff mehrere Tupel aus der Datenbank in die Programmumgebung übertragen werden. Einem Hostarray lässt sich ein entsprechendes Indikatorarray zuordnen. Hostarrays können insbesondere dann verwendet werden, wenn die Größe der Antwortmenge im voraus bekannt ist oder nur ein bestimmter Teil der Antwortmenge interessiert, etwa die ersten 10 Antworten.

```
EXEC SQL BEGIN DECLARE SECTION;
   char cityName[25][20];    /* host array */
   int cityPop[20];         /* host array */
EXEC SQL END DECLARE SECTION;
...
EXEC SQL SELECT Name, Population
         INTO :cityName, :cityPop
         FROM City
         WHERE Code = 'D';
```

Hier wird angenommen, dass die Zahl der deutschen Städte in der Datenbank gleich oder kleiner 20 ist. Somit ist die Verwendung eines Cursors hier nicht nötig. Ist die Zahl der Antworttupel kleiner als die Größe des Hostarrays, so kann die genaue Zahl der übertragenen Tupel über `sqlerrd[2]` in Erfahrung gebracht werden. Reicht die Größe des Array nicht aus um die gesamte Antwortmenge zu fassen, werden nur die ersten 20 Tupel in das Hostarray übertragen. Durch Verwendung eines Cursors ist es jedoch möglich, diesen Schritt so oft zu wiederholen, bis alle Tupel verarbeitet wurden:

```
EXEC SQL BEGIN DECLARE SECTION;
   char cityName[25][20];    /* output host variable */
   int cityPop[20];         /* output host variable */
EXEC SQL END DECLARE SECTION;
```

```
EXEC SQL DECLARE CityCursor CURSOR FOR
   SELECT Name, Population
   FROM City
   WHERE Code = 'D';
EXEC SQL OPEN CityCursor;

EXEC SQL WHENEVER NOT FOUND DO break;
for(;;)
{
   EXEC SQL FETCH CityCursor INTO :cityName, :cityPop;
   /* Verarbeitung der Tupel */
};
```

In jedem Schleifendurchlauf werden hier 20 Tupel in die Arrays übertragen, außer beim letzten Durchlauf, bei dem es ggf. weniger sein können.

`CURRENT OF`-Klausel. Mit der `CURRENT OF <cursor-name>`-Klausel kann in einem `DELETE`- oder `UPDATE`-Statement Bezug auf das Tupel genommen werden, auf das mit dem vorhergehenden `FETCH` zugegriffen wurde.

```
EXEC SQL DECLARE CityCursor CURSOR FOR
     SELECT Name, Population FROM City;
...
EXEC SQL OPEN CityCursor;
EXEC SQL WHENEVER NOT FOUND DO break;
for (;;) {
    EXEC SQL FETCH CityCursor INTO :cityName, :cityPop;
    ...
    EXEC SQL UPDATE City SET Population = :newPop;
    WHERE CURRENT OF CityCursor;
}
```

PL/SQL. Der ORACLE Pro*C/C++ Precompiler unterstützt die Verwendung von PL/SQL-Blöcken (siehe Abschnitt
Neben dem Vorteil, dass PL/SQL direkt in ORACLE integriert ist, bietet es im Zusammenhang mit der Embedded-Programmierung auch einen Laufzeitvorteil gegenüber "gewöhnlichen" SQL-Anweisungen. SQL-Statements werden einzeln an den ORACLE Server gesendet und dort verarbeitet. SQL-Anweisungen die in einem PL/SQL-Block zusammengefasst sind, werden hingegen als Ganzes an den Server geschickt (und so wie eine einzige Anweisung behandelt). Somit kann der Kommunikationsaufwand zwischen Client und Server gesenkt werden. Eingebettete PL/SQL-Blöcke werden in einem Rahmen an die Datenbank übergeben:

```
EXEC SQL EXECUTE
DECLARE
  ...
BEGIN
  ...
```

```
END;
END-EXEC;
```

Transaktionen. Falls ein Anwendungsprogramm nicht durch `COMMIT`- oder `ROLLBACK`-Anweisungen unterteilt ist, wird es als eine geschlossene Transaktion behandelt. In ORACLE wird nach Beendigung des Programms automatisch ein `ROLLBACK` ausgeführt. Falls Änderungen nicht durch ein explizites oder implizites `COMMIT` dauerhaft wurden, gehen diese verloren. DDL-Anweisungen generieren vor und nach ihrer Ausführung implizit ein `COMMIT`. Die Verbindung zur Datenbank sollte "ordentlich" durch

```
EXEC SQL COMMIT RELEASE;    oder
EXEC SQL ROLLBACK RELEASE;
```

beendet werden. Ohne `RELEASE` können nach Beendigung des Anwendungsprogramms noch solange Ressourcen (wie Locks und Cursor) in der Datenbank belegt sein, bis das Datenbanksystem von sich aus erkennt, daß die Verbindung beendet wurde.
Durch Savepoints lassen sich umfangreichere Transaktionen in einzelne Teile gliedern. Mit

```
EXEC SQL SAVEPOINT <name>;
```

wird ein Savepoint gesetzt. Um einen Teil einer Transaktion rückgängig zu machen, verwendet man ein `ROLLBACK` bis zum betreffenden Savepoint, hier

```
EXEC SQL ROLLBACK TO SAVEPOINT <name>;
```

Zusätzlich werden dabei alle Savepoints und Locks nach dem spezifizierten Savepoint gelöscht bzw. freigegeben. Dagegen hat ein `ROLLBACK` ohne Bezug zu einem Savepoint folgende Auswirkungen:

- alle Änderungen der Transaktion werden rückgängig gemacht,
- alle Savepoints werden gelöscht,
- die Transaktion wird beendet,
- alle expliziten Cursore werden gelöscht.

Da ein `ROLLBACK` Teil einer Ausnahmebehandlung sein kann, ist dessen Einsatz auch in Fehlerroutinen sinnvoll.

Fehlerbehandlung. Ein Anwendungsprogramm sollte so konzipiert sein, dass auf "vorhersehbare" Fehlersituationen, die durch ein SQL-Statement entstehen können, noch sinnvoll reagiert werden kann, etwa durch Ausgabe von Fehlermeldungen oder Abbruch einer Schleife. Dazu stehen zwei Mechanismen zur Verfügung: die *SQL Communications Area* (SQLCA) und das `WHENEVER`-Statement. SQLCA ist eine Datenstruktur, die Statusinformationen bzgl. der zuletzt ausgeführten SQL-Anweisung enthält:

```
struct sqlca {
   char   sqlcaid[8];   /* enthaelt den string "SQLCA" */
   long   sqlcabc;      /* die laenge der struktur in bytes */
   long   sqlcode;      /* enthaelt fehlernummer */
   struct {
          unsigned short sqlerrml;  /* laenge des meldungstextes */
          char sqlerrmc[70];       /* meldungstext */
```

```
    } sqlerrm;
    char   sqlerrp[8];   /* zur zeit nicht belegt */
    long   sqlerrd[6];   /* statuscodes */
    char   sqlwarn[8];   /* flags bzgl. warnungen */
    char   sqlext[8];    /* zur zeit nicht belegt */
};
```

Von besonderem Interesse in dieser Datenstruktur ist die Komponente `sqlcode`. Dort sind drei Fälle zu unterscheiden:

· 0: die Verarbeitung einer Anweisung erfolgte ohne Probleme.

· >0: die Verarbeitung ist zwar erfolgt, dabei ist jedoch eine Warnung aufgetreten,
z. B. "no data found".

· <0: es trat ein ernsthafter Fehler auf und die Anweisung konnte nicht ausgeführt werden. In diesem Fall sollte eine entsprechende Fehlerroutine abgearbeitet werden.

Im Array `sqlerrd` sind in der aktuellen Version die dritte Komponente (Anzahl der erfolgreich verarbeiteten Sätze) und die fünfte Komponente (Offset in der SQL-Anweisung, in der ein Übersetzungsfehler aufgetreten ist) belegt. Die weiteren Komponenten sind für zukünftigen Gebrauch reserviert. Die SQLCA wird durch die Headerdatei `sqlca.h` deklariert (siehe demo1.pc).
Mit dem `WHENEVER`-Statement können Aktionen spezifiziert werden, die im Fehlerfall automatisch ausgeführt werden sollen. Die Syntax des `WHENEVER`-Statements lautet:

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

Als `<condition>` sind erlaubt:

· `SQLWARNING` : die letzte Anweisung verursachte eine Warnung (siehe auch `sqlwarn`), die Warnung ist verschieden zu "no data found". Dies entspricht `sqlcode` > 0, aber ungleich 1403.

· `SQLERROR` : die letzte Anweisung verursachte einen (ernsthaften) Fehler (siehe oben). Dies entspricht `sqlcode` < 0.

· `NOT FOUND` : `SELECT INTO` bzw `FETCH` liefern keine Antworttupel zurück. Dies entspricht `sqlcode` 1403.

Als `<action>` kommen in Frage:

· `CONTINUE` : das Programm fährt mit der nächsten Anweisung fort.

· `DO <proc_name>` : Aufruf einer Prozedur (Fehlerroutine); `DO break` zum Abbruch einer Schleife.

· `GOTO <label>` : Sprung zu dem angegebenen Label.

· `STOP` : das Programm wird ohne `commit` beendet (exit()), stattdessen wird ein `rollback` ausgeführt.

Der Geltungsbereich einer `WHENEVER`-Anweisung erstreckt sich bis zur nächsten `WHENEVER`-Anweisung, die bzgl. der `<condition>` übereinstimmt.
Es wird empfohlen, nach jeder relevanten SQL-Anweisung den Status abzufragen, um gegebenenfalls eine Fehlerbehandlung einzuleiten. Der Precompiler

erzeugt automatisch nach jeder SQL-Anweisung eine entsprechende Abfrage gemäß dem aktuellen `WHENEVER`-Statement.[2]

[2] Man betrachte den C-Code, den der Precompiler aus den `WHENEVER`-Statements generiert.

```
/* demo1.pc */

#include <stdio.h>
#include <sqlca.h>

void sql_error() {
   printf("%s \n",sqlca.sqlerrm.sqlerrmc);
   /* in abhaengigkeit von sqlca.sqlcode weitere Anweisungen
      evtl. exit(1) */
}

int main() {

   EXEC SQL BEGIN DECLARE SECTION;
      char cityName[25];    /* Output Host-variable */
      int cityPop;         /* Output Host-variable */
      char* the_code = "D";    /* Input Host-variable */
      short ind1, ind2;      /* Indikator-variablen */
      char* uid = "/";       /* Anmelden ueber Benutzeraccount */
   EXEC SQL END DECLARE SECTION;
   EXEC SQL WHENEVER SQLERROR DO sql_error();
   /* Verbindung zur Datenbank herstellen */
   EXEC SQL CONNECT :uid;

   /* Cursor deklarieren */
   EXEC SQL DECLARE CityCursor CURSOR FOR
      SELECT Name, Population
      FROM City
      WHERE Code = :the_code;
   EXEC SQL OPEN CityCursor;  /* Cursor oeffnen */

   printf("Stadt                        Einwohner\n");
   printf("--------------------------------------\n");

   EXEC SQL WHENEVER NOT FOUND DO break;
   while (1) {
      EXEC SQL FETCH CityCursor INTO :cityName:ind1 ,
                                     :cityPop INDICATOR :ind2;
      if(ind1 != -1 && ind2 != -1)
      {  /* keine Nullwerte ausgeben */
         printf("%s     %d \n", cityName, cityPop);
      }
   };
   EXEC SQL CLOSE CityCursor;
   EXEC SQL ROLLBACK RELEASE;
}
```

Abbildung 17.3: Beispielprogramm

18 JDBC

Die Programmiersprache Java zeichnet sich insbesondere durch ihre Plattformunabhängigkeit aus – auf jedem Computer, auf dem eine *Virtual Java Machine* läuft, können Java-Programme ablaufen. Um Datenbankzugriffe in Java zu standardisieren (und damit die Entwicklung verschiedener, zueinander inkompatibler Dialekte zu verhindern) wurde das JDBC-API[1] (API = Application Programming Interface, eine Sammlung von Klassen und Schnittstellen, die eine bestimmte Funktionalität bereitstellen) entwickelt. JDBC zielt weiterhin darauf ab, eine Anwendung *unabhängig von dem darunterliegenden Datenbanksystem* programmieren zu können. Um dies gewährleisten zu können (JDBC-intern müssen die Statements schließlich über einen Treiber doch wieder in Statements der jeweiligen SQL-Version umgesetzt werden), müssen die verwendeten SQL-Anweisungen dem SQL92 Entry Level entsprechen – ansonsten riskiert man, nur bestimmte Datenbanken ansprechen zu können.
JDBC gilt als "low-level"-Interface, da SQL-Anweisungen als Zeichenketten direkt ausgeführt werden. Auf Basis des JDBC-API lassen sich dann "higher-level"-Anwendungen erstellen. Aufsetzend auf JDBC sind zwei Arten von "higher-level"-APIs in der Entwicklung:

- Embedded SQL (siehe Abschnitt

Ein ähnliches Ziel wie JDBC verfolgt bereits Microsofts ODBC (Open DataBase Connectivity), das eine C-Schnittstelle anbietet, mit der auf fast alle gebräuchlichen Datenbanksysteme zugegriffen werden kann. Derzeit (1999) ist ODBC das am häufigsten verwendete Interface für den Zugriff auf entfernte Datenbanken aus einem Anwendungsprogramm. ODBC kann von Java aus über eine sogenannte JDBC-ODBC-Bridge verwendet werden. Gegenüber ODBC soll JDBC leichter zu verstehen und anzuwenden sein, außerdem werden die bekannten Vorteile von Java bezüglich Sicherheit und Robustheit genutzt. Prinzipiell kann man sich JDBC als ein ODBC vorstellen, das in eine objekt-orientierte Sichtweise übersetzt worden ist. Wie ODBC basiert auch JDBC auf dem X/Open SQL CLI (Call Level Interface) Standard.

18.1 Architektur

Der Zugriff mit JDBC auf die Datenbank erfolgt über (zu JDBC gehörende) Treiber, die mit den spezifischen DBMS, die angesprochen werden sollen, kommunizieren können. Die SQL-Anweisungen werden in den entsprechenden SQL-Dialekt übersetzt und an die Datenbank gesendet, und die Ergebnisse werden zurückgegeben. Um dieses zu ermöglichen, sind

[1] das Akronym für Java Database Connectivity ist zugleich ein eingetragenes Warenzeichen von JavaSoft.

1. geeignete Treiber für das darunterliegende Datenbanksystem, sowie
2. geeignete SQL-Befehle in dem SQL-Dialekt des darunterliegenden Datenbanksystems

erforderlich.
(1) ist hierbei ein reines Programmierproblem, das bei entsprechendem Bedarf und finanziellem Aufwand zu lösen ist, während (b) von den Datenbankherstellern gelöst werden muss.

18.1.1 Treiber

Der Kern von JDBC ist ein *Treibermanager*, der die Java-Anwendungen mit einem geeigneten JDBC-Treiber verbindet, der den Zugriff auf ein Datenbanksystem ermöglicht. Für jedes verwendete Datenbanksystem muss ein solcher JDBC-Treiber installiert sein. Diese JDBC-Treiber können auf unterschiedliche Arten verwirklicht sein:

- Herstellerabhängiges DBMS-Netzwerk-Protokoll mit *pure Java*-Treiber: Der Treiber wandelt JDBC-Aufrufe in das vom DBMS verwendete Netzwerkprotokoll um. Damit kann der DBMS-Server direkt von (JDBC-)Client-Rechner aufgerufen werden. Diese Treiber werden von den DBMS-Herstellern passend zu dem DBMS-Netzwerkprotokoll entwickelt.
- JDBC-Netz mit *pure Java*-Treiber: JDBC-Aufrufe werden in ein von dem/den DBMS unabhängiges Netzprotokoll übersetzt, das dann auf einem Server in ein bestimmtes DBMS-Protokoll übersetzt wird. Über diese Netzserver-Middleware können Java-Clients mit vielen verschiedenen Datenbanken verbunden werden. Es ist somit die flexibelste Lösung – die allerdings auf geeignete Produkte der Hersteller angewiesen ist.

Diese Treiber sind natürlich der bevorzugte Weg, um aus JDBC auf Datenbanken zuzugreifen (u. a., da auch hier die Installation Java-typisch automatisch aus einem Applet erfolgen kann, das vor Benutzung den (pure-Java!)-Treiber lädt). Bis alle entsprechenden Treiber auf dem Markt sind, finden jedoch einige Übergangslösungen Anwendung, die auf bereits vorhandenen (Non-Java)-Treibern basieren:

- JDBC-ODBC-Bridge und ODBC-Treiber: Über die JDBC-ODBC-Bridge werden die ODBC-Treiber verwendet. Dazu muss im allgemeinen der ODBC-Binärcode und häufig auch der Datenbankcode der Clients auf jeder Client-Maschine geladen sein – jede Maschine muss damit ein vollwertiger Client sein.
- Natives API und teilweise in Java geschriebene Treiber: Die JDBC-Aufrufe werden in Aufrufe von Client-APIs der entsprechenden Datenbankhersteller übersetzt. Auch hier wird der Binärcode auf jeder Client-Maschine geladen.

Welche JDBC-Treiber verfügbar sind, kann man aktuell auf `http://www.javasoft.com/products/jdbc` erfahren. Mittlerweile sollten für alle gängigen Datenbanksysteme geeignete pure-Java-Treiber vorhanden sein. Die Tauglichkeit von JDBC-Treibern wird durch eine JDBC-Treiber-Testsuite überprüft und durch die Bezeichnung *JDBC Compliant* bestätigt. Solche Treiber unterstützen zumindest ANSI SQL-2 Entry Level (der SQL-92 Standard).

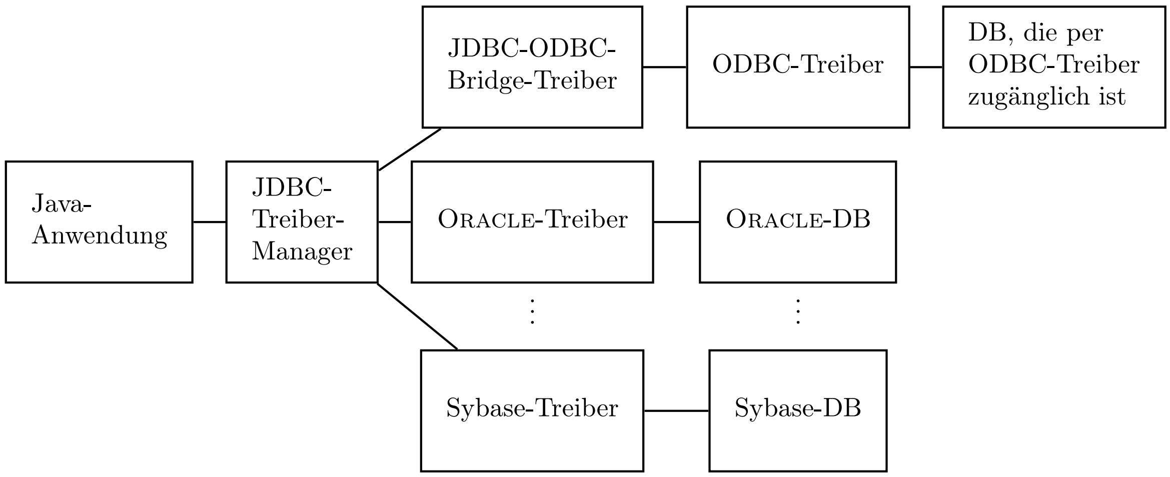

Abbildung 18.1: JDBC-Architektur

Die Architektur von JDBC ist in Abbildung
Der im Praktikum für ORACLE verwendete Treiber gehört zu der zuerst genannten Art: Der Treiber ruft *direkt* den ORACLE-Server auf. Damit sind prinzipiell alle SQL-Befehle von Oracle möglich, man unterliegt also nicht der Einschränkung auf den SQL-2 Standard. Andererseits können Programme, die dies ausnutzen (z. B. Erzeugung von *stored Procedures* oder objektrelationalen Features) i.a. *nicht* mit anderen Datenbanksystemen eingesetzt werden.

18.2 JDBC-Befehle

Die Funktionalität von JDBC lässt sich in drei Bereiche untergliedern:

- Aufbau einer Verbindung zur Datenbank.
- Versenden von SQL-Anweisungen an die Datenbank.
- Verarbeitung der Ergebnismenge.

Für den Benutzer wird diese Funktionalität im wesentlichen durch die im folgenden beschriebenen Klassen `Connection, Statement` (und davon abgeleitete Klassen) und `ResultSet` repräsentiert.

18.2.1 Verbindungsaufbau

Die Verwaltung von Treibern und der Aufbau von Verbindungen wird über die Klasse `DriverManager` abgewickelt. Alle Methoden von `Driver Manager` sind als *static* deklariert – d.h. operieren auf der Klasse, nicht auf irgendwelchen Instanzen. Der Konstruktor für `Driver Manager` ist sogar als `private` deklariert, womit der Benutzer keine Instanzen der Klasse erzeugen kann.
Bevor man eine Verbindung zu einer Datenbank herstellen kann, muss der entsprechende Treiber geladen werden – was hier über den Aufruf der Methode `registerDriver` der Klasse `DriverManager` geschieht:

```
DriverManager.registerDriver(<driver-name>);
```

Welcher Klassenname `<driver-name>` eingesetzt werden muss, steht in dem Handbuch des entsprechenden Treibers, im Praktikum ist dies

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

Der Verbindungsaufbau zu der lokalen Datenbank wird über die Methode

```
DriverManager.getConnection(<jdbc-url>, <user-id>, <passwd>)
```

der Klasse DriverManager vorgenommen. Analog zu den URLs (*Uniform Resource Locator*) des Internet bezeichnet eine JDBC-URL eindeutig eine Datenbank. Eine JDBC-URL besteht wiederum aus drei Teilen:

```
<protocol>:<subprotocol>:<subname>
```

Innerhalb einer JDBC-URL wird immer das `jdbc`-Protokoll verwendet. Dabei bezeichnet `<subprotocol>` den Treiber oder den Datenbank-Zugriffsmechanismus, den ein Treiber unterstützt (siehe Dokumentation des Treibers). Die Datenbank wird durch `<subname>` identifiziert. Der Aufbau des `<subname>` erfolgt in Abhängigkeit des `<subprotocol>`. Für den im Praktikum verwendeten Treiber lautet die Syntax:

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

wobei `<driver-name>` der entsprechende durch `DriverManager.registerDriver(<driver-name>)` registrierte Treibername ist. Wird die User-Id und das Passwort innerhalb der JDBC-URL angegeben, so kann auf die Angabe dieser Argumente innerhalb von `getConnection` verzichtet werden.
Durch den Aufruf von

```
DriverManager.getConnection(<jdbc-url>, <user-id>, <passwd>)
```

wird für jeden registrierten Treiber `<driver>` die Methode `<driver>.connect(<jdbc-url>)` aufgerufen. Derjenige Treiber, der als erster die gewünschte Verbindung herstellen kann, wird genommen. Der Aufruf liefert eine offene Verbindung zurück, die natürlich einem Bezeichner zugewiesen werden muss. Im Beispiel wird die Verbindung `conn` mit

```
String url = “jdbc:oracle:thin:jdbc_1/jdbc_1@132.230.150.11:1521:dev”;
Connection conn = DriverManager.getConnection(url,“dummy”,“passwd”);
```

aufgebaut. Mit der Verbindung an sich kann man eigentlich noch nicht so richtig etwas anfangen. Insbesondere ist es nicht – wie man vielleicht erwartet hätte – möglich, über die Verbindung direkt SQL-Statements an die Datenbank zu übergeben.
Mit der Methode `close` kann ein Connection-Objekt geschlossen werden.

18.2.2 Versenden von SQL-Anweisungen

SQL-Anweisungen werden über `Statement`-Objekte an die Datenbank gesendet. `Statement`-Objekte werden durch Aufruf der Methode `createStatement` (und verwandter Methoden) einer bestehenden Verbindung `<connection>` erzeugt.
Zur Zeit existieren drei verschiedene Klassen von SQL-Anweisungen:

- `Statement`: Instanzen dieser Klasse werden per `<connection>.createStatement()` erzeugt. Mit `Statement` können nur einfache SQL-Anweisungen ohne Parameter verarbeitet werden.
- `PreparedStatement`: Instanzen dieser Klasse werden per `<connection>.prepareStatement(<string>)` erzeugt. Die Klasse `PreparedStatement` ist von der Klasse `Statement` abge-

leitet. Neben der Vorcompilierung von Anfragen erlaubt sie auch die Verwendung von Anfragen mit Parametern (vgl. Abschnitt
Die erzeugte Statement-Instanz wird einem Bezeichner zugewiesen, um später wieder verwendet werden zu können:

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

Die damit erzeugte Instanz `<name>` der Klasse `Statement` kann nun verwendet werden, um SQL-Befehle an die Datenbank zu übermitteln. Dies geschieht abhängig von der Art des SQL-Statements über verschiedene Methoden. Im folgenden bezeichnet `<string>` ein SQL-Statement *ohne Semikolon*.

- `<statement>.executeQuery(<string>)`: Mit `executeQuery` werden *Anfragen* an die Datenbank übergeben. Dabei wird eine Ergebnismenge an eine geeignete Instanz der Klasse `ResultSet` zurückgegeben (siehe Abschnitt
- `<statement>.execute(<string>)`: Die Methode `execute` wird verwendet, wenn ein Statement mehr als eine Ergebnismenge zurückliefert. Dies ist z. B. der Fall, wenn verschiedene gespeicherte Prozeduren und SQL-Befehle nacheinander ausgeführt werden (vgl. Abschnitt

Ein einmal erzeugtes `Statement`-Objekt kann beliebig oft wiederverwendet werden, um SQL-Anweisungen zu übermitteln. Da dabei mit der Übermittlung des nächsten Statements der Objektzustand verändert wird, sind die Übertragungsdaten des vorhergehenden Statements, z. B. Warnungen, danach nicht mehr zugreifbar.
Mit der Methode `close` kann ein Statement-Objekt geschlossen werden.

18.2.3 Behandlung von Ergebnismengen

Die Ergebnismenge des SQL-Statements wird durch eine Instanz der Klasse `ResultSet` repräsentiert:

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

Prinzipiell ist das (wieder einmal) eine virtuelle Tabelle, auf die von der "Hostsprache" – hier also Java – zugegriffen werden kann. Zu diesem Zweck unterhält ein `ResultSet`-Objekt einen Cursor, der durch die Methode `<result-set>.next` jeweils auf das nächste Tupel gesetzt wird. Der Cursor bleibt solange gültig wie die entsprechenden `ResultSet` und `Statement` Objekte existieren. Sind alle Elemente eines ResultSets gelesen, liefert `<result-set>.next` den Wert `false` zurück.
Auf die einzelnen Spalten des jeweils unter dem Cursor befindlichen Ergebnistupels wird über die Methoden `<result-set>.get<type>(<attribute>)` zugegriffen. `<type>` ist dabei ein Java-Datentyp, `<attribute>` kann entweder über den Attributnamen, oder durch die Spaltennummer (beginnend bei 1) gegeben sein. JDBC steht zwischen Java (mit den bekannten Objekttypen) und den verschiedenen SQL-Dialekten (die prinzipiell immer dieselben Datentypen, aber unter unterschiedlichen Namen verwenden). In `java.sql.types` werden *generische* SQL-Typen definiert, mit denen JDBC arbeitet.
Bei dem Aufruf von `get<type>` werden die Daten des Ergebnistupels (die als SQL-Datentypen vorliegen) in Java-Typen konvertiert wie in Abbildung
Die empfohlene Zuordnung der wichtigsten SQL-Typen zu `get<type>`-Methoden ist in Abbildung

Java-Typ	JDBC/SQL-Typ mit Codenummer
`String`	`CHAR (1), VARCHAR (12), LONGVARCHAR (-1)`
`java.math.BigDecimal`	`NUMERIC (2), DECIMAL (3)`
`boolean`	`BIT (-7)`
`byte`	`TINYINT (-6)`
`short`	`SMALLINT (5)`
`int`	`INTEGER (4)`
`long`	`BIGINT (-5)`
`float`	`REAL (7), FLOAT (6)`
`double`	`DOUBLE (8)`
`java.sql.Date`	`DATE (91)` (für ORACLE-Typ `DATE`: Tag, Monat, Jahr)
`java.sql.Time`	`TIME (92)` (für ORACLE-Typ `DATE`: Stunde, Minute, Sekunde)
`java.sql.Timestamp`	`TIMESTAMP (93)`

Abbildung 18.2: Abbildung von Java- auf SQL-Datentypen

Java-Typ	get-Methode
`INTEGER`	`getInt`
`REAL, FLOAT`	`getFloat`
`BIT`	`getBoolean`
`CHAR, VARCHAR`	`getString`
`DATE`	`getDate`
`TIME`	`getTime`

Abbildung 18.3: `get<type>`-Methoden zu JDBC/SQL-Datentypen

Beispiel 41 Ein einfaches Beispiel, das für alle Städte den Namen und die Einwohnerzahl ausgibt, sieht dann etwa so aus:

```
import jdbc.sql.*;

class Hello {
  public static void main (String args []) throws SQLException {
    // \Oracle-Treiber laden
    DriverManager.registerDriver(new oracle.jdbc.dnlddriver.OracleDriver());
    // Verbindung zur Datenbank herstellen
    String url = "jdbc:oracle:dnldthin:@132.230.150.161:1521:test";
    Connection conn = DriverManager.getConnection(url,"dummy","passwd");
    // Anfrage an die Datenbank
    Statement stmt = conn.createStatement ();
    ResultSet rset = stmt.executeQuery("SELECT Name, Population FROM City);
    while (rset.next ()) {   // Verarbeitung der Ergebnismenge
        String s = rset.getString(1);
        int i = rset.getInt("Population");
        System.out.println(s + "  " + i "\n");
    }
  }
```

```
}
```

An dieser Stelle ist anzumerken, dass System.out.println(...) die Ausgabe in das xterm sendet. Für die Anwendung als Applet muss output.appendText(...) verwendet werden (siehe Aufgaben); □

Informationen über die Spalten der Ergebnismenge können mit <result-set>.getMetaData in Erfahrung gebracht werden, womit ein ResultSetMetaData-Objekt erzeugt werden kann, das alle Informationen über die Ergebnismenge enthält:

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

Die wichtigsten Methoden von ResultSetMetaData sind in Abschnitt
Da die Java-Datentypen keine NULL-Werte vorsehen, kann mit der Methode <resultSet>.wasNULL() getestet werden, ob der zuletzt gelesene Spaltenwert NULL war.

Beispiel 42 Das folgende Codefragment gibt die aktuelle Zeile eines Result-Sets – rset einschließlich Nullwerten – aus: ResultSet.getColumnCount gibt die Anzahl der Spalten, die dann mit ResultSet.getString eingelesen werden.

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

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

18.2.4 Prepared Statements

Im Gegensatz zu der Klasse Statement wird bei PreparedStatement die SQL-Anweisung bei der Erzeugung des Statement-Objektes durch

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

vorcompiliert. Im Gegensatz zu einem Statement – bei dem dem bei execute... immer das auszuführende Statement als <string> angegeben wird, ist bei PreparedStatement der SQL-Befehl bereits in dem Objektzustand fest enthalten. Damit ist die Verwendung eines PreparedStatement effizienter als Statement, wenn ein SQL-Statement häufig ausgeführt werden soll.
Abhängig von der Art des SQL-Befehls ist für ein PreparedStatement auch nur eine der (hier parameterlosen!) Methoden <prepared-statement>.executeQuery(), <prepared-statement>.executeUpdate() oder <prepared-statement>.execute() anwendbar, um es auszuführen.
Zusätzlich kann der SQL-Befehl <string> bei der Erzeugung eines PreparedStatements Eingabeparameter, bezeichnet durch "?", enthalten:

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

Die Werte für die “?”-Parameter werden mit Hilfe der 2-stelligen Methoden

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

gesetzt, bevor ein `PreparedStatement` ausgeführt wird. `<type>` gibt dabei den Java-Datentyp an, `<pos>` gibt die Position des zu setzenden Parameters an, `<value>` den zu setzenden Wert.

Beispiel 43 Mit dem folgenden Codefragment können zuerst alle Städte in Deutschland und danach alle Städte in der Schweiz bearbeitet werden:

```
PreparedStatement pstmt = con.prepareStatement("SELECT Population FROM
                                                Country WHERE Code = ?");
pstmt.setString(1,"D");
ResultSet rset = pstmt.ExecuteQuery();
...
pstmt.setString(1,"CH");
rset = pstmt.ExecuteQuery();
...
```

`PreparedStatements` mit Parametern sind insbesondere praktisch, um in Schleifen verwendet zu werden. □

Analog zu `<result-set>.get<type>` bildet Java den Java-Datentyp `<type>` auf einen SQL-Datentyp ab.
Nullwerte können über die Methode `setNULL(<pos>,<type>)` als Parameter angegeben werden, wobei `<type>` den SQL-Typ dieser Spalte bezeichnet:

```
pstmt.setNULL(1,Types.String);
```

18.2.5 Callable Statements: Gespeicherte Prozeduren

Prozeduren und Funktionen werden in der üblichen Syntax (`CREATE PROCEDURE` bzw. `CREATE FUNCTION`) durch

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

als Datenbankobjekte erzeugt. Dann wird der *Aufruf der Prozedur* als `CallableStatement`-Objekt erzeugt. Da die Aufrufsyntax der verschiedenen Datenbanksysteme unterschiedlich ist, wird in JDBC eine generische Syntax per Escape-Sequenz verwendet. Mit dem Befehl

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

wird ein `callableStatement`-Objekt erzeugt. Je nachdem, ob die Prozedur eine Anfrage (die ein *ResultSet* zurückgibt), eine Veränderung an der Datenbank, oder mehrere aufeinanderfolgende SQL-Statements, die unterschiedliche Ergebnisse zurückgeben, ausführt, wird die Prozedur mit

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

oder

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

aufgerufen. Da `CallableStatement` eine Subklasse von `PreparedStatement` ist, können auch hier Parameter – hier auch als `OUT`-Parameter der Prozedur – verwendet werden:

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

Gibt die Prozedur einen Rückgabewert zurück [testen, ob das mit Oracle-PL/SQL-Funktionen klappt], ist die Syntax folgendermaßen:

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

Ob die einzelnen "?"-Parameter `IN`, `OUT`, oder `INOUT`-Parameter sind, hängt von der Prozedurdefinition ab und wird von JDBC anhand der Metadaten der Datenbank selbständig analysiert.
`IN`-Parameter werden wie bei `PreparedStatement` über `set<type>` gesetzt. Für `OUT`-Parameter muss vor der Verwendung zuerst der JDBC-Datentyp der Parameter mit

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

registriert werden (die Prozedurdefinition enthält ja nur den SQL-Datentyp). Um den `OUT`-Parameter dann schlussendlich nach Ausführung der Prozedur lesen zu können, wird die entsprechende `get<type>`-Methode verwendet:

```
<java-type> <var> = <callable-statement>.get<type>(<pos>);
```

Für `INOUT`-Parameter muss ebenfalls `registerOutParameter` aufgerufen werden. Sie werden entsprechend mit `set<type>` gesetzt und mit `get<type>` gelesen.
Liefert der Aufruf eines `CallableStatement`-Objekts mehrere Result Sets zurück, sollte man aus Portabilitätsgründen alle diese Ergebnisse lesen, bevor `OUT`-Parameter gelesen werden.

18.2.6 Sequenzielle Ausführung

Mit `<statement>.execute(<string>)`, `<prepared-statement>.execute()` und `<callable-statement>.execute()`, können SQL-Statements, die mehrere Ergebnismengen zurückliefern, an eine Datenbank übermittelt werden. Dies kommt speziell dann vor, wenn ein SQL-Statement in Java dynamisch als String generiert und dann komplett an die Datenbank gesendet wird (Verwendung der Klasse `Statement`) und man nicht weiß, wieviele Ergebnismengen im speziellen Fall zurückgeliefert werden.
Nach Ausführung von `execute` kann mit `getResultSet` bzw. `getUpdateCount` die erste Ergebnismenge bzw. der erste Update-Zähler entgegengenommen werden. Um jede weitere Ergebnismengen zu holen, muss `getMoreResults` und dann wieder `getResultSet` bzw. `getUpdateCount` aufgerufen werden. Wenn man nicht weiß, welcher Art das erste/nächste Ergebnis ist, muss man es ausprobieren:

- Aufruf von `getResultSet`: Falls das nächste Ergebnis eine Ergebnismenge ist, wird diese zurückgegeben. Falls entweder kein nächstes Ergebnis mehr vorhanden ist, oder das nächste Ergebnis keine Ergebnismenge sondern ein Update-Zähler ist, so wird `null` zurückgegeben.

- Aufruf von `getUpdateCount`: Falls da nächste Ergebnis ein Update-Zähler ist, wird dieser (eine Zahl $n \geq 0$) zurückgegeben; falls das nächste Ergebnis eine Ergebnismenge ist, oder keine weiteren Ergebnisse mehr vorliegen, wird -1 zurückgegeben.
- Aufruf von `getMoreResults`: `true`, wenn das nächste Ergebnis eine Ergebnismenge ist,`false`, wenn es ein Update-Zähler ist, oder keine weiteren Ergebnisse mehr abzuholen sind.
- es sind also alle Ergebnisse verarbeitet, wenn

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

  bzw.

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

  ist.

Dabei kann man mit `getResultSet`, `getUpdateCount` und `getMoreResults` bestimmen, ob ein Ergebnis ein `ResultSet`-Objekt oder ein Update-Zähler ist und eine Ergebnismenge nach der anderen ansprechen.

Beispiel 44 (Sequenzielle Verarbeitung von Ergebnismengen) Das folgende Codesegment (aus [HCF98] geht eine Folge von Ergebnisse (Ergebnismengen und Update-Zähler in beliebiger Reihenfolge) durch:

```
stmt.execute(queryStringWithUnknownResults);
while (true) {
    int rowCount = stmt.getUpdateCount();
    if (rowCount > 0) {
       System.out.println("Rows changed = " + count);
       stmt.getMoreResults();
       continue;
    }
    if (rowCount == 0) {
       System.out.println("No rows changed");
       stmt.getMoreResults();
       continue;
    }
    ResultSet rs = stmt.getResultSet;
    if (rs != null) {
       .....  // benutze Metadaten ueber die Ergebnismenge
       while (rs.next()) {
           ....  // verarbeite Ergebnismenge
           stmt.getMoreResults();
           continue;
    }
    break;
}
```

18.3 Transaktionen in JDBC

Nach der Erzeugung befinden sich Connection-Objekte im *Auto-Commit-Modus*, d.h. alle SQL-Anweisungen werden automatisch als individuelle Transaktionen freigegeben. Der Auto-Commit-Modus kann durch die Methode

```
<connection>.set-AutoCommit(false);
```

aufgehoben werden. Danach können Transaktionen durch

`<connection>.commit()` oder `<connection>.rollback()`

explizit freigegeben oder rückgängig gemacht werden.

18.4 Schnittstellen der JDBC-Klassen

Dieser Abschnitt gibt eine Auflistung relevanter Schnittstellen einzelner JDBC Klassen. Die getroffene Einschränkung erklärt sich daraus, dass für Standardanwendungen nur ein (kleiner) Teil der Methoden benötigt wird.[2]
Die Schnittstellen der JDBC-Klassen Numeric, DatabaseMetaData, Date, DataTruncation, DriverPropertyInfo, CallableStatement, Timestamp, SQLException, SQLWarning, NULLData, Time, Date und Types werden hier deshalb nicht dargestellt.

Driver:

Methode	Beschreibung
boolean acceptsURL(String)	`true` falls Treiber mit angegebener JDBC-URL eine Verbindung zur Datenbank herstellen kann
Connection connect(String,Properties)	Verbindungsaufbau zur Datenbank
boolean jdbcCompliant()	ist es oder ist es nicht?

DriverManager:

Methode	Beschreibung
void registerDriver(Driver)	Treiber wird dem DriverManager bekannt gemacht
Connection getConnection(String)	Verbindungsaufbau
Connection getConnection(String, Properties)	Verbindungsaufbau
Connection getConnection(String, String, String)	Verbindungsaufbau
Driver getDriver(String)	Versucht, einen Treiber zu finden, der die durch String gegebene URL versteht
Enumeration getDrivers()	Liste der verfügbaren JDBC-Treiber
void deregisterDriver(Driver)	
DriverManager DriverManager()	Konstruktor (`private`)

Connection:

[2]Die Klasse `DatabaseMetaData` verfügt über mehr als 100 Schnittstellen.

Methode	Beschreibung
Statement createStatement()	s.o.
CallableSatement prepareCall(String)	s.o.
PreparedStatement prepareStatement(String)	s.o.
DatabaseMetaData getMetaData()	eine Instanz der Klasse DatabaseMetaData enthält Informationen über die Verbindung
void commit()	alle Änderungen seit dem letzten Commit bzw. Rollback werden dauerhaft, alle Locks werden freigegeben
void rollback()	alle Änderungen seit dem letzten Commit bzw. Rollback werden verworfen, alle Locks werden freigegeben
void clearWarnings()	alle Warnings der Verbindung werden gelöscht
SQLWarning getWarnings()	erste Warnung die im Zusammenhang mit der Verbindung aufgetreten ist
void setAutoCommit(boolean)	auto-commit Status wird gesetzt
void setReadOnly(boolean)	read-only Modus wird spezifiziert
boolean isReadOnly()	
void close()	die Verbindung zwischen der Instanz und der Datenbank wird beendet
boolean isClosed()	

Statement:

Methode	Beschreibung
ResultSet executeQuery(String)	s.o.
int executeUpdate(String)	s.o.
boolean execute(String)	s.o
ResultSet getResultSet()	Ergebnismenge zum ausgeführten SQL Statement
int getUpdateCount()	Ergebnis des letzten Update
void clearWarnings()	alle Warnings der Statement-Instanz werden gelöscht
SQLWarning getWarnings()	erste Warnung im Zusammenhang mit dem Statement

PreparedStatement: Subklasse von Statement

Methode	Beschreibung
ResultSet executeQuery()	s.o.
int executeUpdate()	s.o.
boolean execute()	s.o.
void set‹type›(int, ‹type›)	entsprechend dem Parameterindex wird ein Argument vom Datentyp ‹type› gesetzt

ResultSet:

Methode	Beschreibung
⟨type⟩ get⟨type⟩(int)	gibt eine Instanz vom Typ ⟨type⟩ entsprechend dem Spaltenindex zurück
⟨type⟩ get⟨type⟩(String)	gibt eine Instanz vom Typ ⟨type⟩ entsprechend dem Attributnamen zurück
ResultSetMetaData getMetaData()	eine Instanz der Klasse ResultSetMetaData enthält Informationen über Anzahl, Typ und weitere Eigenschaften der Spalten der Ergebnismenge
int findColumn(String)	Spaltenindex zum Attributnamen
boolean next()	es wird auf das nächste Ergebnistupel zugegriffen, Rückgabewert ist `false` wenn keine weiteren Ergebnistupel existieren
boolean wasNull()	`true` falls der zu letzt gelesene Wert `NULL` war, sonst false
void close()	Verbindungsabbau zwischen der ResultSet Instanz und Datenbank

ResultSetMetaData:

Methode	Beschreibung
int getColumnCount()	Spaltenanzahl der Ergebnismenge
String getColumnLabel(int)	Attributname der Spalte ⟨int⟩
String getTableName(int)	Tabellenname der Spalte ⟨int⟩
String getSchemaName(int)	Schemaname der Spalte ⟨int⟩
int getColumnType(int)	JDBC-Typ der Spalte ⟨int⟩ (als Integer-Wert, siehe Abbildung
Unterliegender DBMS-Typ der Spalte ⟨int⟩	