Embedded SQL, JDBC

Kopplungsarten zwischen Datenbank- und Programmiersprachen

- Erweiterung der Datenbanksprache um Programmierkonstrukte (z.B. PL/SQL)
- Erweiterung von Programmiersprachen um Datenbankkonstrukte: Persistente Programmiersprachen, Datenbank-Programmiersprachen
- Datenbankzugriff aus einer Programmiersprache
- Einbettung der Datenbanksprache in eine Programmiersprache: “Embedded SQL”

Embedded SQL

- C, Pascal, C++

Impedance Mismatch bei der SQL-Einbettung

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

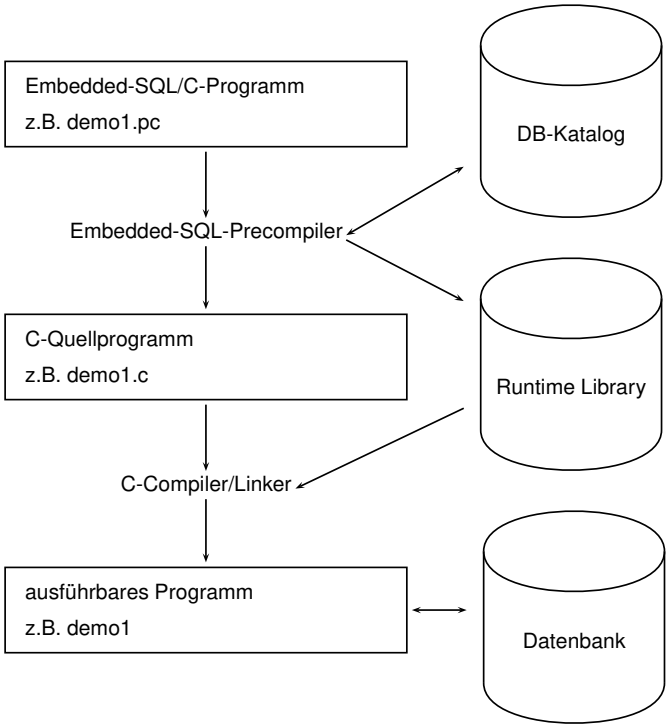
Realisierte Lösung

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

Auswirkungen auf die Hostsprache

- Struktur der Hostsprache bleibt unverändert
- Jede SQL-Anweisung kann eingebettet werden
- SQL-Anweisungen wird `EXEC SQL` vorangestellt
- Wie kommunizieren Anwendungsprogramm und Datenbank ?

Entwicklung einer Embedded SQL-Applikation



Verbindungsaufbau

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

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

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

Äquivalent:

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

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

Hostvariablen

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

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

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

Indikatorvariablen

Verarbeitung von Nullwerten

Indikatorvariablen für Output-Variablen:

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

Indikatorvariablen für Input-Variablen:

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

Cursore

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

Cursor-Operationen

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

Fehlersituationen

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

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

Beispiel

```
int main() {
   EXEC SQL BEGIN DECLARE SECTION;
      char cityName[25];    /* output host var */
      int cityEinw;         /* output host var */
      char* landID = "D";    /* input host var */
      short ind1, ind2;      /* indicator var */
      char* uid = "/";
   EXEC SQL END DECLARE SECTION;
   /* Verbindung zur Datenbank herstellen */
   EXEC SQL CONNECT :uid;
   /* Cursor deklarieren */
   EXEC SQL DECLARE StadtCursor CURSOR FOR
      SELECT Name, Einwohner
      FROM Stadt
      WHERE Code = :landID;
   EXEC SQL OPEN StadtCursor;  /* Cursor oeffnen */
   printf("Stadt                  Einwohner\n");
   while (1)
   {EXEC SQL FETCH StadtCursor INTO :cityName:ind1 ,
                         :cityEinw INDICATOR :ind2;
    if(ind1 != -1 && ind2 != -1)
    {  /* keine NULLwerte ausgeben */
       printf("%s     %d \n", cityName, cityEinw);
    }};
   EXEC SQL CLOSE StadtCursor; }
```

Hostarrays

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

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

holt 20 Tupel in die beiden Hostarrays.

PL/SQL

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

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

Statisches vs. Dynamisches SQL

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

Transaktionen

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

Savepoints

- Transaktion läßt sich durch Savepoints unterteilen.
- Syntax : `EXEC SQL SAVEPOINT <name>`
- Ein `ROLLBACK` zu einem Savepoint löscht alle danach gesetzten Savepoints.

Mechanismen für Ausnahmebehandlung

- SQL Communications Area (SQLCA)
- `WHENEVER`-Statement

SQLCA

enthält Statusinformationen bzgl. der zuletzt ausgeführten SQL-Anweisung

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

Interpretation der Komponente `sqlcode`:

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

WHENEVER-Statement

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

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

<condition>

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

<action>

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

Java und Datenbanken

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

JDBC: Datenbankzugriff-API

- Interface für den (entfernten) Datenbankzugriff von Java-Programmen aus
- Applikation kann unabhängig vom darunterliegenden DBMS programmiert werden
- setzt die Idee von ODBC auf Java um
- gemeinsame Grundlage ist der X/Open SQL CLI (Call Level Interface) Standard

JDBC-Architektur

JDBC-Architektur

- Kern: Treiber-Manager
- darunter: Treiber für einzelne DBMS'e

Arten von Treibern:

- Angestrebt:
  - DBMS-Client-Server-Netzwerk-Protokoll mit *pure Java*-Treiber: JDBC-Aufrufe werden in das DBMS-Netzwerkprotokoll umgesetzt. JDBC-Client ruft direkt den DBMS-Server auf.
  - JDBC-Netz mit *pure Java*-Treiber: JDBC-Aufrufe werden in das JDBC-Netzwerkprotokoll übersetzt, und dann auf einem Server in ein bestimmtes DBMS-Protokoll übersetzt.
- als Übergangslösungen:
  - JDBC-ODBC-Bridge und ODBC-Treiber: Über die JDBC-ODBC-Bridge werden die ODBC-Treiber verwendet.
  - Natives API: Die JDBC-Aufrufe werden in Aufrufe von Client-APIs der entsprechenden Datenbankhersteller übersetzt.

JDBC-API

- flexibel:
  - Applikation kann unabhängig vom darunterliegenden DBMS programmiert werden
  - de facto: Portabilität nur im SQL-2 Standard gesichert (stored procedures, objektrelationale Features)
- “low-level”:
  - Statements werden durch Strings übertragen
  - im Gegensatz zu Embedded SQL keine Verwendung von Programmvariablen in den SQL-Befehlen

Darauf aufbauend in Entwicklung:

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

JDBC-Funktionalität

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

JDBC-Treiber-Manager

`DriverManager`

- verwaltet (registriert) Treiber
- wählt bei Verbindungswunsch den passenden Treiber aus und stellt Verbindung zur Datenbank her.

- Es wird nur ein `DriverManager` benötigt.

⇒ Klasse `DriverManager`:

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

Benötigte Treiber müssen angemeldet werden:

`DriverManager.registerDriver(driver*)`

Im Praktikum für den Oracle-Treiber:

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

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

Verbindungsaufbau

- Aufruf an den `DriverManager`:

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

- Datenbank wird eindeutig durch JDBC-URL bezeichnet

JDBC-URL:

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

Bei uns:

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

```
String url =
  'jdbc:oracle:thin:@132.230.150.11:1521:dev';
Connection conn =
  DriverManager.getConnection(url,'jdbc_1','jdbc_1');
```

liefert eine offene Verbindungs-Instanz `conn` zurück.

Verbindung beenden: `conn.close();`

Versenden von SQL-Anweisungen

`Statement`-Objekte

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

Klasse “Statement”

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

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

- `ResultSet <statement>.executeQuery(<string>)`: *Anfragen* an die Datenbank. Dabei wird eine Ergebnismenge zurückgegeben.
- `int <statement>.executeUpdate(<string>)`: SQL-Statements, die eine Veränderung an der Datenbasis vornehmen. Der Rückgabewert gibt an, wieviele Tupel von der SQL-Anweisung betroffen waren.
- `<statement>.execute(<string>)`: Statement gibt mehr als eine Ergebnismenge zurück (Folge von Anweisungen). Ergebnismengen etc. sind ueber das Statement-Objekt abfragbar (siehe später).

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

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

Behandlung von Ergebnismengen

Klasse "`ResultSet`":

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

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

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

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

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

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

Behandlung von Ergebnismengen

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

  `<result-set>.get<type>(<attribute>)`
- `<type>` ist dabei ein Java-Datentyp,

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

  `<getString>` funktioniert immer.
- `<attribute>` kann entweder durch Attributnamen, oder durch die Spaltennummer gegeben sein.

  ```
  countries.getString(``Code'');
  countries.getInt(``Population'');
  countries.getInt(3);
  ```
- Bei `get<type>` werden die Daten des Ergebnistupels (SQL-Datentypen) in Java-Typen konvertiert.

Behandlung von Ergebnismengen

```
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,:username,: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");
   }
  }
}
```


Arbeiten mit Ergebnismengen

JDBC-Datentypen

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

Java-Typ	JDBC-SQL-Typ
`String`	`CHAR, VARCHAR`
`java.math.BigDecimal`	`NUMBER, NUMERIC, DECIMAL`
`boolean`	`BIT`
`byte`	`TINYINT`
`short`	`SMALLINT`
`int`	`INTEGER`
`long`	`BIGINT`
`float`	`REAL`
`double`	`FLOAT, DOUBLE`
`java.sql.Date`	`DATE` (Tag, Monat, Jahr)
`java.sql.Time`	`TIME` (Stunde, Minute, Sekunde)

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

Behandlung von Ergebnismengen

Informationen über die Spalten der Ergebnismenge:

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

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

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>
String getColumnTypeName(int)	Unterliegender DBMS-Typ der Spalte <int>

Behandlung von Ergebnismengen

- keine `NULL`-Werte in Java:

  `<resultSet>.wasNULL()`

  testet, ob der zuletzt gelesene Spaltenwert `NULL` war.

Beispiel: Ausgabe der aktuellen Zeile eines ResultSets

```
ResultSetMetaData rsetmetadata = rset.getMetaData();
int numCols = rsetmetadata.getColumnCount();
for(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.

Prepared Statements

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

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

  anwendbar.

Prepared Statements: Parameter

- Eingabeparameter werden durch “?” repräsentiert

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

- “?”-Parameter werden mit

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

  gesetzt, bevor ein `PreparedStatement` ausgeführt wird.
- `<type>`: Java-Datentyp,
- `<pos>`: Position des zu setzenden Parameters,
- `<value>`: Wert.

```
pstmt.setString(1,"D");
ResultSet rset = pstmt.ExecuteQuery();
...
pstmt.setString(1,"CH");
ResultSet rset = pstmt.ExecuteQuery();
...
```

- Nullwerte werden durch

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

  gesetzt, wobei `<type>` den JDBC-Typ dieser Spalte bezeichnet:

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

Callable Statements: Gespeicherte Prozeduren

- Erzeugen von Prozeduren und Funktionen mit

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

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

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

- der *Aufruf der Prozedur* wird als `CallableStatement`-Objekt erzeugt:
- Aufrufsyntax von Prozeduren bei den verschiedenen Datenbanksystemen unterschiedlich

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

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


Callable Statements mit Parametern

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

- Parameter:

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

- Rückgabewert bei Funktionen:

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

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

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

registriert werden.

```
cstmt.registerOutParameter(1,java.sql.types.number);
```

Callable Statements mit Parametern

- Vorbereitung (s.o.)

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

- IN-Parameter werden über set<type> gesetzt.

```
cstmt.setString(2,'Freiburg');
cstmt.setString(3,'Berlin');
```

- Aufruf mit

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

oder

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

oder

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

Im Beispiel: `cstmt.execute();`

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

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


Sequentielle Ausführung

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

Sequentielle Ausführung

- `getResultSet()`: Falls nächstes Ergebnis eine Ergebnismenge ist, wird diese zurückgegeben; falls kein nächstes Ergebnis mehr vorhanden, oder nächstes Ergebnis ein Update-Zähler ist: `null` zurückgeben.
- `getUpdateCount()`: Falls nächste Ergebnis ein Update-Zähler ist, wird dieser ($n \geq 0$) zurückgegeben; falls kein nächstes Ergebnis mehr vorhanden, oder nächstes Ergebnis eine Ergebnismenge ist, wird -1 zurückgegeben.
- `getMoreResults()`: `true`, wenn das nächste Ergebnis eine Ergebnismenge ist, `false`, wenn es ein Update-Zähler ist, oder keine weiteren Ergebnisse.
- alle Ergebnisse verarbeitet:

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

bzw.

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


Folge von Ergebnissen verarbeiten

```
stmt.execute(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) {
       .....  // verarbeite Metadaten
       while (rs.next())
          { ....}  // verarbeite Ergebnismenge
       stmt.getMoreResults();
       continue;
    }
    break;
}
```

Weitere SQL/Oracle-Werkzeuge

- Dynamic SQL: SQL-Statements können *in PL-SQL* bei Laufzeit als Strings zusammengesetzt und an die Datenbank übermittelt werden.
- ORACLE8i: Mit eingebauter Java Virtual Machine, Zugriff auf das Filesystem,
  i= internet: XML-Schnittstelle, Web-Application-Server etc.
- ORACLE-Web Server/Internet Application Server (9i): HTML-Seiten werden mit abhängig vom Datenbankinhalt erstellt.
- mit den neuesten Paketen (IAS, Internet File System Server) verwischt der Unterschied zwischen Datenbank und Betriebssystem.