3. Zugriffe auf RDB mit JDBC (Java Database Connectivity)

Der Zugriff auf relationale Datenbanken mittels Java ist durch den Sprachstandard JDBC (=: Java Database Connectivity) geregelt. Der Standard unterscheidet zwischen Spezifika einzelner DBMS-Produkte und der allgemeinen Verarbeitung von SQL-Befehlen in Java. Die Spezifika kommen in der Hauptsache nur beim **Verbindungsaufbau (Connection)** zum Tragen. Die **allgemeine Verarbeitung von SQL-Befehlen in Java** ist unabhängig von DBMS-Produktspezifika und wird mittels des Java-Pakets **java.sql** implementiert. Dieses Paket enthält Schnittstellen, Klassen und Ausnahmefälle.

<Abb.1: Allgemeine Übersicht: JDBC-Zugriff>

Legende:

- **JDB1** **:** Java-Clientprogramm mit JDBC-Zugriff auf eine RDB.
- **T1, …, Tn** **:** Tabellen der RDB. Das RDBMS läuft auf einem Server-Rechner.
- **connect_A** **:** Verbindungsanfrage.
- **connection** **:** Verbindungsantwort (eine Instanz der Klasse Connection).
- **SQL_A** **:** Statement mit SQL-Anfrage.
- **SQL_E** : Ergebnis der SQL-Anfrage (SQL-Antwort).

Das allgemeine Programmierkonzept ist daher: Der Verbindungsaufbau wird pro DBMS in einer **produktspezifischen** connection()-Methode programmiert. Alle weiteren DB-Zugriffe sind produktunabhängig und damit nur von der Logik des Algorithmus und von SQL abhängig und somit **portierbar** auf ein beliebiges RDBMS, das sich an die SQL-Normen hält.

3.1. JDBC-Verbindungsaufbau

RDBMS-Produktanbieter (Oracle, IBM, Microsoft, mySQL, ...) stellen JDBC-Treiber zur Verfügung. Vor dem Kompilieren ist der jeweilige JDBC-Treiber für die Java Entwicklungsumgebung zu laden.

Der Verbindungsaufbau von einem Java Client-Programm arbeitet in zwei Schritten: (1) ein produktspezifischer Treiber wird geladen, (2) mittels des Treibers wird unter Angabe einer URL beim RDBMS Server-Rechner eine Verbindungsinstanz (eine Instanz der Klasse **Connection**) angefordert. Im Erfolgsfall wird eine Verbindungsinstanz zurückgegeben. Im Fehlerfall wird eine SQLException erzeugt.

Nachfolgend ist eine Oracle spezifische Methode **connect()** beispielhaft angegeben. Im Falle eines anderen RDBMS müsste nur diese Methode durch eine andere connect() Methode ausgetauscht werden. Alle anderen Klassen und Methoden eines JDBC-Programms bleiben unverändert. Die im nachfolgenden Beispiel enthaltene Klasse OracleDataSet ist im produktspezifischen Paket **oracle.jdbc.pool** enthalten, das importiert werden muss.

BSP.1: Eine Oracle spezifische Methode **connect():**

```
/*****************************************************************/
/* Methode     : connect()                                       */
/* Zweck       : Aufbau einer Oracle spezifischen DB-Verbindung  */
/* Parameter   : KEINE                                           */
/* Rückgabewert: Instanz der Klasse Connection                   */
/* Exception(s): SQLException                                    */
/*****************************************************************/
public static Connection connect() throws SQLException
{String treiber;
 OracleDataSource ods    = new OracleDataSource();
 treiber = "oracle.jdbc.driver.OracleDriver";
 Connection dbConnection = null;
 /* Treiber laden                    */
 try
 {Class.forName(treiber).newInstance();
 } catch (Exception e)
   {System.out.println("Fehler beim laden des Treibers: "+ e.getMessage());
   }
 /* Datenbank-Verbindung erstellen    */
 try
 {ods.setURL("jdbc:oracle:thin:MMMM/KKKK@//Boetius.nt.fh-koeln.de:TTTT:xe");
  dbConnection = ods.getConnection();
 } catch (SQLException e)
   { System.out.println("Fehler beim Verbindungsaufbau zur Datenbank!");
     System.out.println(e.getMessage());
   }
 return dbConnection;
}
```

3.2 SQL- und Java-Datentypen

Die folgende Tabelle gibt es eine Übersicht, welche SQL-Datentypen welchen Java-Datentypen entsprechen.

SQL-Datentypen	**Java-Datentypen**
integer	int
float	double
decimal(p,q)	double (in Annäherung), BigDecimal (Stellengenau)
char(n), varchar(n)	String
date, time, timestamp	Date (Weiterverarbeitung mit Calendar)

3.3 Erzeugung eines Anweisungsobjekts

Für jede SQL-Anweisung, die in einem JDBC-Programm ausgeführt werden soll, muß ein Anweisungsobjekt, d.h. eine Instanz der Klasse **Statement** erzeugt werden. Hierfür benötigt man eine Instanz der Klasse **Connection**.

BSP.2: Erzeugung eines Anweisungsobjekts. Das dabei benutzte Objekt **dbConnection** ist eine bereits existierende Instanz der Klasse **Connection**:

```
Statement st1;
st1 = dbConnection.createStatement( );
```

3.4 SELECT unter JDBC

Die Ausführung eines SELECT Kommandos und die Verbeitung seines Ergebnisses unter JDBC besteht in der Hauptsache aus vier Arbeitsschritten:

(1) Aufbau eines SELECT-Strings
(2) Ausführung der SELECT-Anfrage
(3) Verarbeitung der Ergebnismenge
(4) Schließen des Anweisungsobjekts

(1) Aufbau des SELECT-Strings
Jedes SQL-Kommando, das von einem JDBC-Programm an den SQL-Kommandointerpreter übergeben wird, ist eine Zeichenkette, d.h. ein Stringobjekt. In den String können anwendungsspezifische Variablen aufgenommen werden.

BSP.3: Ein konstantes SELECT-Kommando, das einen JOIN auf die Tabellen AUTO und KFZMIETV (Kfz-Mietverträge) enthält:

```
String SQ1="SELECT AUTOID, AUTONAME, MIETVID, PID FROM
AUTO A, KFZMIETV B WHERE A.AUTOID=B.AUID";
```

BSP.4: Ein SELECT auf eine Tabelle **Kunde**, wobei die Grenzen des PLZ-Bereiches durch vorgegebene Variablenwerte vom Typ **int** bestimmt sind:

```
String SQ2="SELECT knr, knam, plz, kredit FROM Kunde WHERE
plz >=”+ug+” AND plz <=”+og;
```

BSP.5: Ein SELECT auf eine Tabelle **Kunde**, wobei ein Vergleichsmuster durch eine Stringvariable bestimmt sind:

```
String SQ3="SELECT knr, knam, plz, kredit FROM Kunde WHERE
knam LIKE '"+vgl+"%'";
```

BSP.5: Ein SELECT mit Gruppenverarbeitung auf eine Tabelle **Kunde**, wobei in der Spaltenauswahl ein Aliasname für einen arithmetischen Ausdruck gesetzt wird:

```
String SQ4="SELECT ort, count(knr)AS anzk FROM Kunde GROUP
BY ort";
```

(2) Ausführung der SELECT-Anfrage

Mit einem vorhandenen Anweisungsobjekt **st1** der Klasse Statement (vgl. 3.3) und einem String **sqsel**, der den SELECT-Befehl enthält, wird mittels der Methode **executeQuery()** der Klasse **Statement** die SELECT-Anfrage an das RDBMS gestellt. Die SELECT-Antwort, die eine leere Ergebnisliste (SQLNotFoundException) oder eine Ergebnisliste mit mindestens einer Zeile zurückgibt, wird als Instanz einer Klasse **ResultSet** verwaltet.

Prototyp: public **ResultSet** executeQuery(String sql) throws **SQLException**

BSP.6: Aufruf der Methode executeQuery():

```
    ResultSet rs1;
    rs1 = st1.executeQuery(sqsel);
```

(3) Verarbeitung der Ergebnismenge

Im allgemeinen Fall besteht die Ergebnismenge aus einer oder mehreren Zeilen. Die Ergebnismenge wird in der Regel mittels einer Schleife verarbeitet. Durch die Ergebnismenge kann mit der Methode **next()** der Klasse **ResultSet** navigiert werden. Die Methode next() stellt eine Kombination der üblichen Iterator-Methoden hasNext() und next() für Mengen- bzw. Listenobjekte dar: **next()** prüft, ob eine erste bzw. nächste Ergebniszeile vorliegt. Ist dieses der Fall, wird direkt auf diese Ergebniszeile zugegriffen.

Die Struktur einer Ergebniszeile ist durch die Spaltenauswahl des SELECT gegeben. Auf jeden Eintrag der Spaltenauswahl kann durch Angabe des Spaltennamens **spn** oder durch Angabe der Spaltenummer **k** zugegriffen werden ($1 \leq k \leq n$; n = Anzahl der Spalten in der Spaltenauswahl). Der Zugriff erfolgt mit einer datentypspezifischen Methode **getDta()** der Klasse **ResultSet**, wenn **dta** ein für die Verarbeitung der Ergebnisspalte geeigneter Java Datentyp ist (vgl. 3.2).

Allgemeine Prototypen der get-Methoden der Klasse ResultSet:

```
dta getDta(String spn)
dta getDta(int k)
```

Java-Datentypen	**getDta() Methoden**
int	getInt()
double	getDouble()

BigDecimal	getBigDecimal()
String	getString()
Date	getDate()

Ist die Ergebnismenge verarbeitet worden, kann der Zugriff darauf mit der Methode **close()** geschlossen werden: **`rs1.close();`**

(4) Schließen des Anweisungsobjekts
Um das Anweisungsobjekt, mit dem die SELECT-Anfrage gestellt wurde freizugeben, wird es mit der Methode **close()** der Klasse Statement geschlossen.

BSP.7: Ausführung der SELECT-Anfrage aus BSP.3 und Verarbeitung der Ergebnismenge:

```
static void kfzAbfrage(){
int iz=0;
try
{ System.out.println("START:");
  Connection con = connect();

  Statement Stmt;
  ResultSet RS;
  String SQL;

  String knam;
  int kid, mvid, mpid;
```

```
  // Erzeugen eines Statements aus der DB-Verbindung
  Stmt = con.createStatement();
/*************************************************************/
/*                  Eine SQL-SELECT Anfrage                  */
/*************************************************************/
  SQL  = "SELECT AUTOID, AUTONAME, MIETVID, PID FROM AUTO A,
     KFZMIETV B WHERE A.AUTOID=B.AUID";

  RS   = Stmt.executeQuery(SQL);
  System.out.println("Ergebnisliste:");

  while(RS.next())
      {knam = RS.getString("AUTONAME");
       kid  = RS.getInt("AUTOID");
       mvid = RS.getInt("MIETVID");
       mpid = RS.getInt("PID");
       System.out.println("KFZID: "+kid+" KFZNAME: "+knam+
                          " MIETVID: "+mvid+" MIETERNR: "+mpid);
       iz++;
      }
  RS.close();
  Stmt.close();
 }
 catch (SQLException e)
```

```
 { System.out.println(e.getMessage());
    System.out.println("SQL Exception wurde geworfen!");
 }
 System.out.println("DB-Abfrage: "+iz+" Zeilen gefunden.");
}
```

BSP.8: Ein SELECT unter JDBC, das die Ergebnisliste des SELECT als typisierte Liste zurückgibt. Die Ergebnisse des SELECT auf eine Tabelle ARTIKEL werden durch Instanzen der folgenden Klasse **`Artikel`** verwaltet:

```
class Artikel
{int artnr;
 String artbez;
 double preis;
 Artikel(int eartnr, String eartbez, double epreis)
 {artnr=eartnr;
  artbez=eartbez;
  preis=epreis;
 }
}
```

Die Methode hat folgenden Aufbau:

```
static LinkedList<Artikel> liesAllArt(Connection c1)
{LinkedList<Artikel> lz=new LinkedList<Artikel>();
 // JDBC Objekte zur Kommunikation mit der DB
```

```
 Statement Stmt;
 ResultSet RS;
 String SQL;
 int aartnr;
 String aartbez;
 double apreis;
 Artikel ax;
 try
 {// Erzeugen eines Statements aus der DB-Verbindung
  Stmt = c1.createStatement();
  /*****************************************************/
  /*         SELECT Anfrage auf alle Artikel           */
  /*****************************************************/
  SQL  = "SELECT artnr,artbez,preis FROM Artikel ORDER BY
artnr";
  // SQL-Anweisung ausführen und Ergebnis in ein ResultSet schreiben
  RS   = Stmt.executeQuery(SQL);
  // Das ResultSet zeilenweise durchlaufen
  while(RS.next())
  {aartbez = RS.getString("artbez");
   aartnr  = RS.getInt("artnr");
   apreis  = RS.getDouble("preis");
```

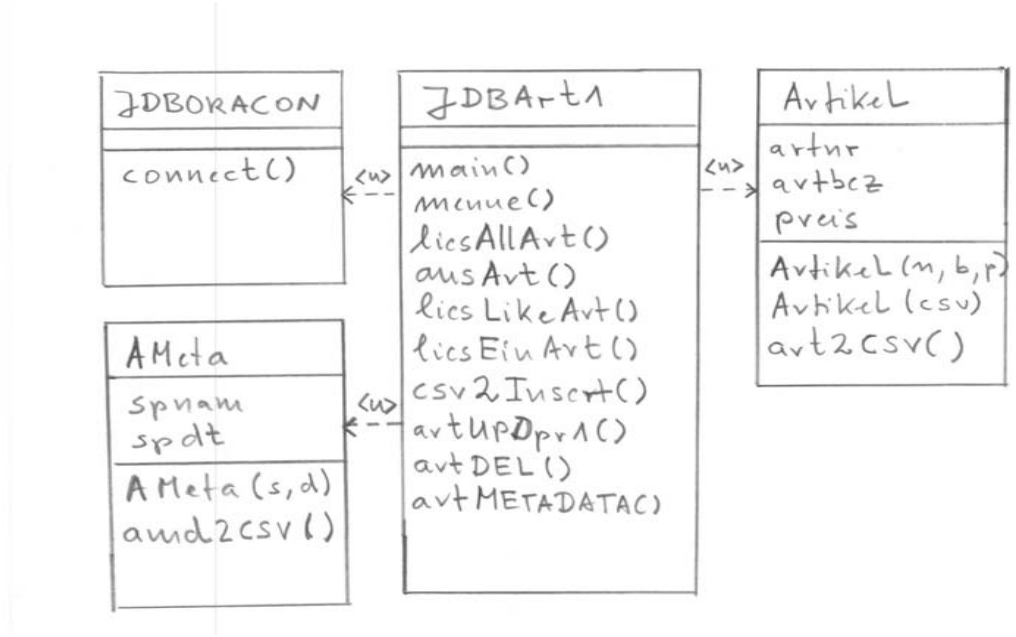
```
 ax=new Artikel(aartnr,aartbez,apreis);
 lz.add(ax);
}
RS.close();
Stmt.close();
// SQL Exception abfangen
} catch (SQLException e){
  System.out.println(e.getMessage());
  System.out.println("SQL Exception wurde geworfen!");
  }
  return lz;
 }
```

3.5 Schreibende SQL-Zugriffe unter JDBC

Die Ausführung eines INSERT, UPDATE und DELETE Kommandos hat ebenso wie die Ausführung eines SELECT ein STATEMENT-Objekt als Träger. Der Hauptunterschied besteht darin, dass statt der Methode executeQuery() für schreibende Zugriffe die Methode **executeUpdate()** der Klasse **Statement** ausgeführt wird.
Prototyp: int executeUpdate(String SQLIUD)

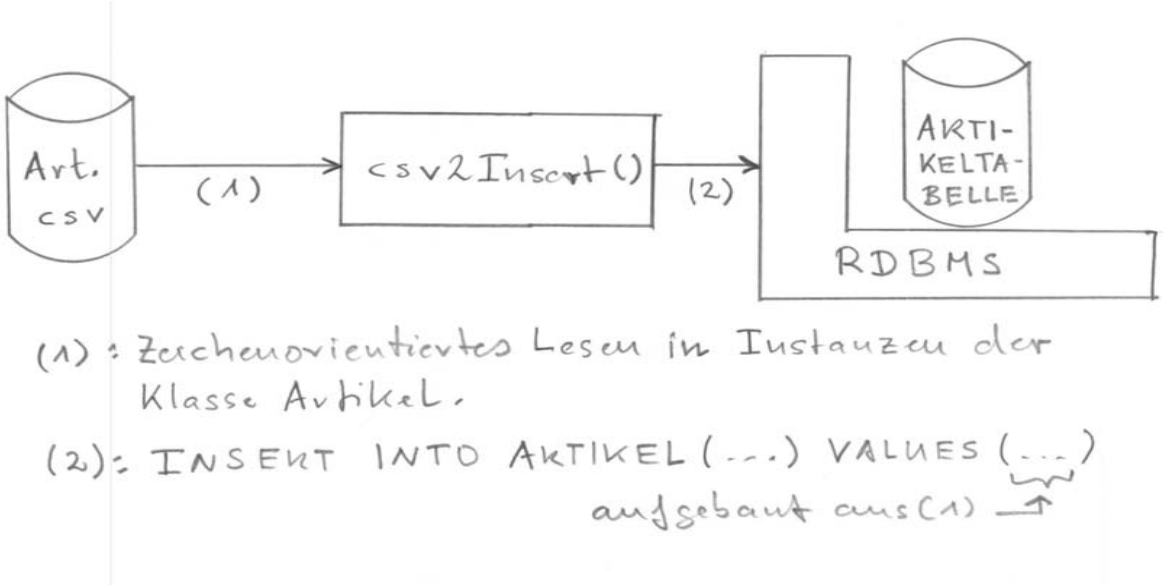
Der String SQLIUD enthält ein INSERT, UPDATE oder DELETE Kommando. Der Rückgabewert ist die Anzahl korrekt eingefügter, überschriebener oder gelöschter Zeilen.

Die nachfolgenden Beispiele für schreibende JDBC Zugriffe sind aus dem im Folgenden dokumentierten Programmsystem übernommen:



<Abb.2: Klassendiagramm>

BSP.9: Im Folgenden wird ein INSERT unter JDBC auf die Artikeltabelle ausgeführt. Dabei wir die VALUES-Klausel aus den Attributwerten einer Artikelinstanz gefüllt. Die Artikelinstanz wurde durch einen speziellen Konstruktor aus einem CSV-Datensatz aufgebaut, der aus einer CSV-Datei zeichenorientiert eingelesen wurde. Diese Verarbeitung wird innerhalb der Methode **csv2Insert()**, deren Datenflussplan nachfolgend gegeben ist, ausgeführt.



<Abb.3:DFP: csv2Insert()>

Für das INSERT unter JDBC sind die folgenden Arbeitsschritte wesentlich:

(1) Deklaration der benötigten JDBC Objekte:

```
int iz=0; Statement Stmt; String SQL; Artikel ax;
```

(2) Aufbau der Instanz, die die VALUES-Klausel mit Werten versorgt. Hier ist die Quelle ein CSV-String h, mit der der CSV-Konstruktor der Klasse Artikel arbeitet:

```
ax=new Artikel(h);
```

(3) Anlegen des Statement-Objekts:

```
Stmt = c1.createStatement();
```

(4) Aufbau des SQL-INSERT Strings:

```
SQL="INSERT INTO Artikel(artnr,artbez,preis)
VALUES("+ax.artnr+",'"+ax.artbez+"',"+ax.preis+")";
```

(5) Ausführen des INSERT unter JDBC:

```
iz = Stmt.executeUpdate(SQL);
```

iz hat in dem Fall, dass dieses INSERT korrekt ausgeführt wurde, den Wert 1, sonst enthält **iz** den Wert 0.

Der Quelltext der Methode **csv2Insert()**:

```
static int csv2Insert(Connection c1, String dsn) throws IOException
   {int r=0, dz=0, iz=0;
    Statement Stmt;
    ResultSet RS;
```

```
String SQL,h;
Artikel ax;

FileReader fr1=new FileReader(dsn);
BufferedReader br1=new BufferedReader(fr1);
h=br1.readLine();
while(h!=null)
{dz=dz+1;
 ax=new Artikel(h);
 if (ax.artnr==-1)
 {System.out.println("Kein INSERT fuer: "+h);
  h=br1.readLine();
  continue;
 }
 try
 {Stmt = c1.createStatement();
/*****************************************************/
/*                  INSERT fuer ein Artikel          */
/*****************************************************/
 SQL="INSERT INTO Artikel(artnr,artbez,preis)
 VALUES("+ax.artnr+",'"+ax.artbez+"',"+ax.preis+")";
  // INSERT ausführen
  iz = Stmt.executeUpdate(SQL);
  r=r+iz;
  Stmt.close();
```

```
     } catch (SQLException e){
       System.out.println(e.getMessage());
       System.out.println("SQL Exception wurde geworfen!");
     }
     h=br1.readLine();
    }
    return r;
   }
```

BSP.10: Im Folgenden können zwei Arten eines UPDATE unter JDBC auf die Artikeltabelle ausgeführt werden: (1) Setzen eines neuen absoluten Preises **epr** für einen Artikel, der durch einen PRIK-Wert **uanr** identifiziert wird. (2) Prozentuale Erhöhung aller Preise. In dem Fall enthält **epr** den Prozentsatz der Preiserhöhung. Im Fall (1) wird in Abhängigkeit von **epr** und **uanr** die SET- und WHERE-Klausel aufgebaut. Im Fall (2) wird nur eine SET-Klausel aufgebaut:

```
SQL="UPDATE Artikel SET preis=";
a=1.+epr/100.;
if (umod==1) SQLE=""+epr+" WHERE artnr="+uanr;
if (umod==2) SQLE=""+a+"*preis";
SQL=SQL+SQLE;
ir = Stmt.executeUpdate(SQL);
```

Beide beschriebene Arten des UPDATE unter JDBC werden mittels der Methode **artUPDpr1()** ausgeführt:

```
static int artUPDpr1(Connection c1, int umod, double epr, int
uanr)
{int ir=0;
 double a=1.;
 Statement Stmt;
 ResultSet RS;
 String SQL, SQLE="";
 try
 {Stmt = c1.createStatement();

/*************************************************************/
/*                  UPDATE fuer einen bzw. mehrere Artikel     */
/*************************************************************/
  SQL="UPDATE Artikel SET preis=";
  a=1.+epr/100.;
  if (umod==1) SQLE=""+epr+" WHERE artnr="+uanr;
  if (umod==2) SQLE=""+a+"*preis";
  SQL=SQL+SQLE;
  ir = Stmt.executeUpdate(SQL);
  Stmt.close();
 } catch (SQLException e){
   System.out.println(e.getMessage());
   System.out.println("SQL Exception wurde geworfen!");
 }
 return ir;
```

```
}
```

Im Fall (2) werden alle Zeilen der Artikeltabelle geändert, daher ist der Rückgabewert **ir** gleich der **Zeilenanzahl** dieser Tabelle.

BSP.11: Genau eine Zeile zu einer übergebenen Artikelnummer **uanr** (Kandidat für einen PRIK Wert) soll gelöscht werden:

```
SQL="DELETE FROM Artikel WHERE artnr="+uanr;
ir = Stmt.executeUpdate(SQL);
```

Dieses DELETE unter JDBC wird mittels der Methode **artDEL()**ausgeführt:

```
static int artDEL(Connection c1, int uanr)
{int ir=0;
 Statement Stmt;
 ResultSet RS;
 String SQL;
 try
 {Stmt = c1.createStatement();
  /*************************************************************/
  /*                  DELETE eines Artikels                    */
  /*************************************************************/
  SQL="DELETE FROM Artikel WHERE artnr="+uanr;
  // DELETE ausführen
  ir = Stmt.executeUpdate(SQL);
  Stmt.close();
```

```
  } catch (SQLException e){
     System.out.println(e.getMessage());
     System.out.println("SQL Exception wurde geworfen!");
  }
  return ir;
 }
```

3.6 Die Abfrage von Metadaten

Unter JDBC besteht die Möglichkeit während der Laufzeit eines Programmes Metadaten einer Tabelle, wie die Spaltenanzahl, die Attributnamen und ihre Datentypen abzufragen. Diese Aktivitäten können mit der Klasse **ResultSetMetaData** ausgeführt werden. Gegeben ist eine Instanz **con1** der Klasse **Connection**. Folgende Schritte sind dann zur Metadatenabfrage auszuführen:

(1) Ein SELECT auf **alle** Spalten der untersuchten Tabelle **tabname** ist auszuführen:
String SQL = “SELECT * FROM tabname“;
Statement st1=con1.createStatement();
ResultSet rs1=st1.executeQuery(SQL);

(2) Eine Metadateninstanz zum ResultSet anlegen:
ResultSetMetaData rsmd1;
rsmd1=rs1.getMetaData();

(3) Spaltenanzahl abfragen:
int m=rsmd1.getColumnCount();

(4) Die Liste der spaltenbezogenen Metadaten aufbauen:
(4a) Einen Knotentyp für die Liste der Metadaten definieren:

```
class AMeta
{String spnam; /* Spaltenname */
 String spdt; /* Datentyp der Spalte */
 …
}
```

(4b) Abfrage des Spaltennamens und seines Datentyps mit folgenden Abfragemethoden der Klasse ResultSetMetadata (Prototypen):

```
String getColumnName(int i)
String getColumnTypeName(int i)
```

Der Übergabeparameter **i** ist der SQL-Spaltenindex (**1 ≤ i ≤ m**).

Die Abfrage der Metadaten bezüglich der Tabelle **Artikel** kann mit folgender Methode **artMETADATA()** ausgeführt werden, die eine Liste von Metadaten für die Spalten dieser Tabelle erzeugt, deren Knoten vom Typ **AMeta** (s. (4a)) sind, erzeugt:

```
static LinkedList<AMeta> artMETADATA(Connection c1)
{LinkedList<AMeta> lz=new LinkedList<AMeta>();
 Statement Stmt;
 ResultSet RS;
```

```
ResultSetMetaData rsmd1;
AMeta x;
String SQL;
int spanz,stanz,i;
String colnam;
String coldtyp;
int nako;

try
{Stmt = c1.createStatement();
 SQL  = "SELECT artnr,artbez,preis FROM Artikel ORDER BY artnr";
 RS   = Stmt.executeQuery(SQL);
 // Metadatenabfrage
 rsmd1=RS.getMetaData();
 spanz=rsmd1.getColumnCount();
 for(i=1;i<=spanz;i++)
 {colnam = rsmd1.getColumnName(i);
  coldtyp= rsmd1.getColumnTypeName(i);
  x=new AMeta(colnam,coldtyp);
  lz.add(x);
 }
 RS.close();
 Stmt.close();
 // SQL Exception abfangen
} catch (SQLException e){
```

```
        System.out.println(e.getMessage());
        System.out.println("SQL Exception wurde geworfen!");
 }
 return lz;
}
```

Für die Tabelle Artikel wird damit folgende Liste von Metadaten erzeugt, deren Knoten in Form von CSV-Strings ausgegeben werden:

```
Relationenschema (JDBC-Abfrage) der Tabelle Artikel:
Spaltenname;Datentyp der Spalte
ARTNR;NUMBER
ARTBEZ;VARCHAR2
PREIS;NUMBER
Ende der Metadaten-Abfrage: 3 Spalteneintraege gefunden.
```

3.7 Verarbeitung von SQL-Sonderdatentypen

3.7.1 Verarbeitung von decimal(p,q)-Attributen mit dem Java Datentyp BigDecimal

Um Spaltenwerte vom SQL-Datentyp **decimal(p,q)** in einem JDBC Programm zu lesen, als Festpunktzahlen sachgerecht zu verarbeiten und wieder in die Datenbank einzufügen, sind folgende Arbeitsschritte auszuführen:

1) Einen decimal(p,q)-Attibutwert aus der DB mit der ResultSet-Methode **getBigDecimal()** lesen. Die Klasse **BigDecimal** ist im Java-Paket **java.math** gegeben. Dieses Paket muss importiert werden. Ein Beispielquelltext zum Lesen lautet:

```
BigDecimal bd1;
bd1=rs1.getBigDecimal("preis");
/* rs1 :  das hier gegebene ResultSet */
```

2) **Konvertierungen:** BigDecimal-Werte können aus Zeichenketten (exakt) und double-Zahlen (in Annäherung) erzeugt werden und auch wieder in Stringwerte und Gleitpunktzahlen konvertiert werden:

a) BigDecimal -> String: String s1=bd1.toString();

b) String -> BigDecimal:
```
String s3=new String("3.1415");
BigDecimal bd3;
bd3=new BigDecimal(s3);
```

c) BigDecimal->double:
```
double x1;
x1=bd1.doubleValue();
```

d) double->BigDecimal:
```
double x3=2.718281;
BigDecimal bd5;
bd5=new BigDecimal(x3);
```

3) **Festpunktarithmetik:**

Addieren: bd3=bd1.add(bd2);
Subtrahieren: bd3=bd1.subtract(bd2);
Multiplizieren: bd3=bd1.multiply(bd2);
Dividieren: bd3=bd1.divide(bd2, int RUNDUNGSMODUS); /* oder */
bd3=bd1.divide(bd2, int nako, int RUNDUNGSMODUS);
/* nako : Anzahl der Nachkommastellen des Quotienten */

Der wichtigste Rundungsmodus ist das kaufmännische Runden: Die Konstante heißt ROUND_HALF_UP und hat den Wert 4.

Mit der **setScale()**-Methode kann eine BigDecimal-Zahl auf q Nachkommastellen gerundet werden: Prototyp: **BigDecimal setScale(int q, int RUNDUNGSMODUS);**

BSP.12: Kaufmännisches Runden von bd3 auf 2 Nachkommastellen:

```
bd3=bd3.setScale(2, ROUND_HALF_UP);
```

Mit der Methode **scale()** kann die Anzahl der Nachkommastellen erfragt werden:

```
int q=bd3.scale();
```

BSP.13: Eine einfache BigDecimal-Rechenmethode:

```
public static BigDecimal bigArith(BigDecimal a, BigDecimal
b, char op)
{BigDecimal c=new BigDecimal(0.0);
  int na, nb, nc, w=1;
  switch(op)
   {case '+': c=a.add(b);
              break;
    case '-': c=a.subtract(b);
              break;
    case '*': c=a.multiply(b);
              break;
    case '/': nc=a.scale()+b.scale();
              c=a.divide(b,nc,4);
              break;
   }
   nc=c.scale();
   System.out.println("c = "+c+" NaKoAnzahl = "+nc);
   System.out.println("Neue Skalierung? (NaKo >= 1 !)");
   nc=IO1.einint();
   c=c.setScale(nc,4);
```

```
   System.out.println("c = "+c+" NaKo = "+nc);
   return c;
 }
```

5) Ein decimal(p,q)-Attibut in der DB mit einem BigDecimal-Wert **überschreiben**: Hierbei ist bd5 ein neu erzeugter BigDecimal-Wert und eartnr eine gegebene Artikelnummer:

```
k=sto1.executeUpdate("UPDATE artikel SET preis="+bd5+"
WHERE artnr="+eartnr);
```

6) Unter JDBC besteht die Möglichkeit während der Laufzeit eines Programmes die **Metadaten** eines **decimal(p,q)**-Attributs abzufragen. Die Zahl **p** ist die Gesamtstellenanzahl der Festpunktzahl und heißt **Precision**. Die Zahl **q** ist die Anzahl der Nachkommastellen und heißt **Scale**. Wenn nun, wie in Kap.3.6, mit **rsmd1** eine Metadateninstanz gegeben ist und **i** die Spaltennummer eines **decimal(p,q)**-Attributs **Ai** ist, kann durch folgende Abfragen **p** und **q** bestimmt werden:

```
    int p=rsmd1.getPrecision(i);
    int q=rsmd1.getScale(i);
```

3.7.2 Verarbeitung von Attributen, die den SQL-Datentyp DATE, TIME oder TIMESTAMP haben

1) **Ein SQL-DATE-Wert aus einem ResultSet rs1 lesen.** In diesem Beispiel hat die Tabelle ARTIKEL, deren Zeilen ins ResultSet **rs1** eingelesen wurden, ein Attribut **bearbdatum**, das vom SQL-Typ DATE ist. Hierzu wird die ResultSet-Methode **getDate()** verwendet, die einen Wert vom Java-Typ Date zurückgibt, der im Paket **java.util** definiert ist:

```
java.util.Date dat1=rs1.getDate("bearbdatum");
```

2) **Konvertierungen**:
a) DATE->String: `String sdat1=dat1.toString();`
b) String->DATE: Hierbei ist Voraussetzung, dass der String das Datum als korrekte SQL-Datumskonstante in der Form JJJJ-MM-TT enthält:

```
String edat=new String("2006-11-26");
java.util.Date dat5= java.util.Date.valueOf(edat);
```

3) **Rechnen mit Datumsbestandteilen**:
Um mit den Datumsbestandteilen wie JJJJ, MM, TT usw. ganzzahlig rechnen zu können, muss ein Date-Wert in eine Kalender-Instanz konvertiert werden:
a) Kalender-Instanz erzeugen:

```
Calendar cal1;
cal1=Calendar.getInstance();
```

b) Kalender-Instanz mit Date-Wert füllen. Hierbei ist dat1 ein Wert vom Typ Date (s.o. 1)): **`cal1.setTime(dat1);`**

c) Kalender-Instanz auswerten:

```
int j1, m1, t1, ms1;
j1=cal1.get(Calendar.YEAR);
m1=cal1.get(Calendar.MONTH);
t1=cal1.get(Calendar.DAY_OF_MONTH);
ms1= cal1.get(Calendar.MILLISECOND);
```

d) Mit Datumsbestandteilen rechnen:
d1) direkt: Z.B. nächstes Jahr: j1=j1+1;
d2) mit Kalender-Methoden: Z.B. 15 Tage nach dem 23.12.2006:

```
Calendar cal15=cal1.add(Calendar. DAY_OF_MONTH,15);
```

Die Rechnung erfolgt mittels Monats- und gfs. Jahreswechsel. Im Beispiel müsste cal5 dann das Datum 7.1.2007 enthalten. Anschließend können, wie in c) die neu errechneten Jahres-, Monats- und Tageswerte in **int**-Variablen wie **j1**, **m1** und **t1** gespeichert werden.

e) Eine Kalender-Instanz in einen String konvertieren:

```
String ex=cal5.toString();
```

4) Ein UPDATE mit einem neu berechneten Datumswert ausführen. Hierbei wird vorausgesetzt, dass wie in 3c) bzw. 3d) die int-Variablen **j1**, **m1** und **t1** die neu errechneten Jahres-, Monats- und Tageswerte enthalten:

```
String SQU="UPDATE Artikel SET bearbdatum=
TO_Date('"+t1+"."+m1+"."+j1+"','DD.MM.YYYY')";
```

5) Ein DATE-Attibut in der DB mit einem SQL-DATE-Wert **du1** überschreiben, wobei hier der **UPDATE** als ein prepared statement ausgeführt werden soll. Hierbei wird der Wert **du1** wie in 2b) als korrekter SQL-Datumswert vorausgesetzt. Im Beispiel soll das Attribut **bearbdatum** für den Artikel mit **artnr=4712** ein neues Datum bekommen.

Das UPDATE-Kommando soll mit einer **PreparedStatement-Instanz** ausgeführt werden. Mit **c** ist hier die existierende DB-Verbindungsinstanz bezeichnet:

a) Ein Prepared Statement anlegen:

```
String prep=new String();
prep="UPDATE ARTIKEL SET bearbdatum = ? WHERE artnr = ?";
PreparedStatement ps=c. prepareStatement(prep);
```

b) Die Platzhalter mit Werten füllen:

```
ps.setInt(2,4712);
 /* edx ist ein String mit einem korrekten SQL-Datumswert:
```

```
    z.B.: edx="2006-11-26" */
java.sql.Date du1= java.sql.Date.valueOf(edx);
ps.setDate(1,du1);
```

c) Das Prepared Statement ausführen:

```
int ku=-3;
ku=ps.executeUpdate();
```

Lernziele zu Kap.3: Zugriffe auf RDB mit JDBC

1. **Die Entkopplung von produktspezifischem Verbindungsaufbau und produktunabhängiger Verarbeitung von SQL-Kommandos unter JDBC verstanden haben.**
2. **Das Vorgehen bei einem JDBC-Zugriff programmieren können: Verbindungen aufbauen, Statement-Objekte anlegen, korrekte Strings mit SQL-Befehlen (SELECT, INSERT, UPDATE, DELETE) schreiben und zugehörige Statements ausführen können (excuteQuery() bzw. executeUpdate()).**
3. **Bei lesenden Zugriffen Ergebnismengen (ResultSets) verarbeiten können.**
4. **Die Durchführung schreibender Zugriffen kontrollieren können.**
5. **Die Metadaten einer Tabelle abfragen können.**
6. **SQL-Sonderdatentypen wie decimal(p,q) und date verarbeiten können.**