JDBC – Kurze Einführung

JDBC ist die Implementierung des SQL/CLI (*SQL Call-Level Interface*) für die Programmiersprache Java. JDBC ist also eine Klassenbibliothek für den Zugriff auf SQL-Datenbanken in Java-Programmen.

Die Bezeichnung JDBC ist offiziell kein Akronym, sondern ein Markenname von Oracle America, Inc., wird aber oft als *Java Database Connectivity* bezeichnet. [Diese Bezeichnung kommt von ODBC (*Open Database Connectivity*), Microsofts Implementierung des SQL/CLI für die Programmiersprache C, bzw. C++.]

Wichtige Ziele von JDBC sind:

- JDBC ist eine Schnittstelle, die die Datenbank-Sprache SQL verwendet.
- JDBC ermöglicht den gleichzeitigen Zugriff auf verschiedene Datenquellen.
- JDBC hat ein „adaptives“ Programmiermodell.
- Einfachheit („Keep it simple“)
- Robustheit, Verfügbarkeit und Skalierbarkeit
- JDBC ist Bestandteil der JSE- und JEE-Architektur von SUN.
- JDBC berücksichtigt die wichtigsten Elemente von SQL:2003.
- JDBC ist die Grundlage für andere Datenbankzugrifsstechniken, wie etwa SQLJ oder die Java Persistence API JPA.

In dieser kurzen Einführung wird an einem grundlegenden Beispiel gezeigt, wie man JDBC verwendet. Die einzelnen Schritte werden dann näher erläutert und dabei werden die wichtigsten Klassen, Interfaces und Methoden der Bibliothek vorgestellt. Details findet man in der Spezifikation von JDBC, siehe http://download.oracle.com/otndocs/jcp/jdbc-4_2-mrel2-spec/index.html und http://www.oracle.com/technetwork/java/javase/jdbc/index.html, sowie in den Tutorien von Oracle http://www.oracle.com/technetwork/java/learning-139577.html.

Das grundlegende Beispiel

```
import java.sql.*;

public class BasicJDBC {

  public static void main(String[] args) {
```

```
    Connection con = null;
    Statement stmt = null;
    ResultSet rs   = null;

    try {

      /** Schritt 1: JDBC-Treiber registrieren */
      Class.forName("org.postgresql.Driver");

      /** Schritt 2: Connection zum Datenbanksystem herstellen */
      con = DriverManager.getConnection(
                     "jdbc:postgresql://localhost/azamon", "dis", "ChrisDate");

      /** Schritt 3: Statement erzeugen */
      stmt = con.createStatement();

      /** Schritt 4: Statement direkt ausfuehren */
      rs = stmt.executeQuery("select author, title from Books");

      /** Schritt 5: Ergebnis der Anfrage verwenden */
      while (rs.next()) {
        System.out.println(rs.getString("author") + " "
                           + rs.getString("title"));
      }
    } catch (Exception e) {
      System.out.println(e.getMessage());
    } finally {

      /** Schritt 6: Ressourcen freigeben */
      try {
        if (rs != null) rs.close();
        if (stmt != null) stmt.close();
        if (con != null) con.close();
      } catch (Exception e) {
        System.out.println(e.getMessage());
      }
    }
  }
}
```

Es sind sechs Schritte, die die Grundstruktur eines JDBC-Programms bilden:

1. **Treiber registrieren**

   ```
   Class.forName("org.postgresql.Driver")
   ```

   Mit der statischen Methode `forName` von `Class` wird die Klasse `Driver` geladen, in unserem Fall der JDBC-Treiber für PostgreSQL. Damit die Klasse vom Classloader von Java wirklich geladen werden kann, muss sich die jar-Datei, die sie enthält, dem Klassenpfad hinzugefügt werden. Ein statischer Block in der Treiberklasse erzeugt ein Objekt des

Treibers und registriert es beim Treibermanager. Der Treiber ist spezifisch für das verwendete Datenbankmanagementsystem, er eröffnet dem Programm den Datenzugriff auf eine Datenquelle.

Seit JDBC Version 4 ist das explizite Laden des Treibers nicht mehr notwendig, denn der Treibermanage lädt automatisch alle JDBC-Treiber, die er auf dem Klassenpfad findet.

2. **Connection zur Datenbank herstellen**

   ```
   Connection con = DriverManager.getConnection(...)
   ```

   Der JDBC-Treibermanager `DriverManager` ist ein Singleton, sein Konstruktor ist privat und alle seine Methoden sind statisch. Er verwendet die JDBC-URL (in unserem Fall `jdbc:postgresql://localhost/azamon)`, bietet sie jedem registrierten Treiber an, bis sich der erste findet, der zu dieser URL eine Connection herstellen kann. Der Treiber instantiiert ein Objekt des Typs `Connection`, das der Treibermanager der Anwendung übergibt. (Danach spielt der Treibermanager gar keine Rolle mehr, es ist das Connection-Objekt, das für die Anwendung die Verbindung zur Datenquelle repräsentiert.)

3. **Statement erzeugen**

   ```
   Statement stmt = con.createStatement()
   ```

   Das Objekt des Typs `Statement` wird von der Connection erzeugt; es wird verwendet, um SQL-Anweisungen an die Datenquelle zu richten.

4. **Anweisung ausführen und Ergebnismenge erzeugen**

   ```
   ResultSet rs = stmt.executeQuery(...)
   ```

   Die SQL-Anweisung wird ausgeführt und die Methode `executeQuery` liefert eine Referenz auf die Ergebnismenge zurück. Die `ResultSet` repräsentiert die Ergebnismenge einer SQL-Anweisung und hat Methoden, mit denen der (implizite) Cursor auf der Ergebnismenge bewegt wird und mit denen die Werte aus der Ergebnismenge gelesen werden. Im einfachsten (und im Default-) Fall hat das Objekt `rs` einen Cursor, der mit `rs.next()` schrittweise vorwärts über die Zeilen des Ergebnisses bewegt werden kann.

5. **Ergebnis verwenden**

   ```
   while ( rs.next() ) {
     ...
   }
   ```

   Die Methode `next()` der `ResultSet` positioniert den Cursor zunächst auf die erste Zeile der Ergebnismenge und bewegt ihn dann mit jeder Iteration über die folgenden Zeilen. Mit den Methoden `getXXX()`, in unserem Fall `getString()`, können die Werte der Spalten aus der aktuellen Zeile der Ergebnismenge gelesen werden. Als Parameter der

Get-Methoden kann man den Namen der Spalte oder auch ihre Position (bei 1 beginnend) angeben. JDBC ist sehr komfortabel was die Konvertierung der Datentypen angeht — in der JDBC-Dokumentation geben Listen Auskunft über die möglichen Konversionen.

6. **Nicht mehr benötigte Resourcen freigeben**

   ```
   rs.close()
   ...
   ```

   Sicherlich würde auch schlussendlich der Garbage-Collector von Java die Objekte vernichten und in diesem Zuge die entsprechende Methode `close()` aufrufen. Auch wenn ein übergeordnetes Objekt vernichtet wird, ruft JDBC die entsprechenden Funktionen der untergeordneten Objekte auf. In diesem Beispiel wird trotzdem die ganze Reihung der Close-Methoden explizit angegeben, um auf folgenden Punkt aufmerksam zu machen: Die Garbage-Collection von Java kann keine Ahnung davon haben, wie *externe* Resourcen verwaltet werden; Resourcen, die die Datenbank bereitstellt. Infolgedessen kann es auch keinen Mechanismus geben, der den geeigneten Zeitpunkt automatisch ermittelt, zu dem diese Resourcen freigegeben werden sollten. Es ist also guter Stil in einem JDBC-Programm, nicht mehr benötigte Resourcen selbst freizugeben.

Architektur von JDBC

Die grundlegende Idee der Architektur besteht darin, die Anwendung vom direkten Zugriff auf das Datenbanksystem durch eine Zwischenschicht zu trennen. Denn der direkte Zugriff würde erfordern, dass die Anwendung Kenntnisse über die spezielle Zugriffstechnik des gerade verwendeten Datenbankmanagementsystems hätte – eine starke Kopplung. JDBC vermindert diese Kopplung, indem die Anwendung nur die JDBC-API verwendet, die (im Prinzip) unabhängig vom DBMS ist, siehe Abb. 1.

JDBC enthält (im Package `java.sql`) den Treibermanager, der JDBC-Treiber verwaltet. Die JDBC-Treiber sind für den eigentlichen Zugriff auf die Datenquelle zuständig, sie sind also spezifisch für die jeweilige Zugriffstechnik und stammen in der Regel vom Anbieter des verwendeten Datenbankmanagementsystems.

Der JDBC-Treiber wird beim JDBC-Treibermanager registriert. Wenn nun die Anwendung eine Connection zur Datenquelle herstellen möchte, ermittelt der Treibermanager aus der URL der Connection, welchen Treiber er verwenden möchte und delegiert die Anfrage an den entsprechenden Treiber weiter. Ab dann werden alle Aufträge an der JDBC-API, die die Anwendung verwendet, an den JDBC-Treiber weitergeleitet.

Der Aufruf von `Class.forName` mit der JDBC-Treiber-Klasse führt zur Re-

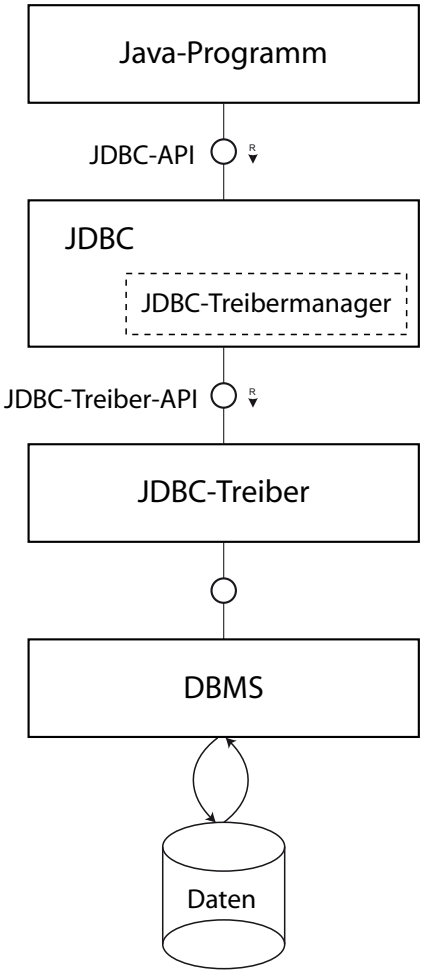

Abbildung 1: Die Architektur von JDBC

gistrierung des Treibers im Treibermanager. Seit JDBC 4.0 kann auf diesen Aufruf verzichtet werden, denn der sogenannte *Server Provider mechanism* von Java lädt JDBC-Treiber automatisch (vorausgesetzt es handelt sich um Treiber, die JDBC 4.0 unterstützen).

Typen von JDBC-Treiber

Es gibt verschiedene Typen von JDBC-Treiber, siehe Abb. 2.

1. **Treiber vom Typ 1** implementieren die JDBC-API durch das Mapping der Funktionen auf eine andere SQL/CLI, die nicht in Java geschrieben ist. In der Regel handelt es sich dabei um ODBC *Open Database Connectivity*, d.h. Treiber vom Typ 1 sind eine JDBC-ODBC-Bridge. Ein solcher Treiber war Bestandteil von SUNs JDK. (Seit Java

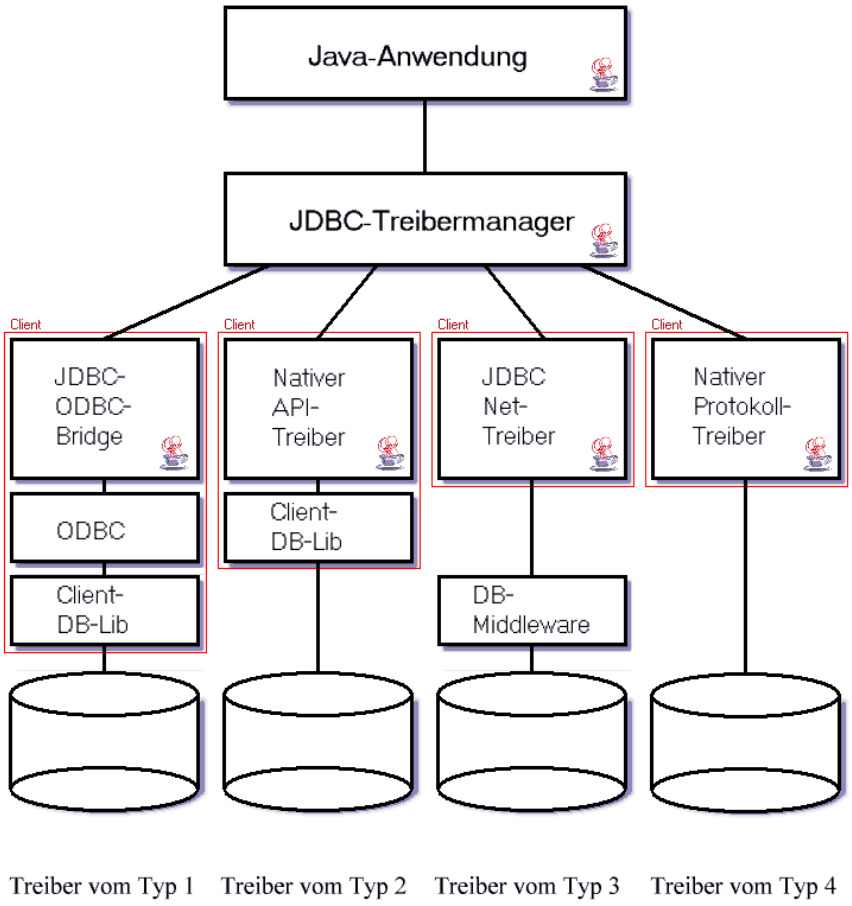

Abbildung 2: Typen von JDBC-Treibern

1.8 ist der JDBC-ODBC-Treiber von Sun nicht mehr Bestandteil des Java JDKs, man findet jedoch im Internet andere Implementierungen dieses Konzepts.) Der Vorteil dieser Technik bestand darin, dass man mit Java vorhandene und installierte ODBC-Funktionalität sofort nutzen konnte — es stand somit von der ersten JDBC-Version an der Zugriff auf nahezu alle denkbaren Datenquellen zur Verfügung. Der Nachteil liegt darin, dass ODBC nicht für Applets genutzt werden kann, weil ODBC die Sicherheitsrestriktionen für Applets verletzt.

2. **Treiber vom Typ 2** beruhen auf den Datenbank-spezifischen Schnittstellen, d.h. sie verwenden die (in der Regel in C geschriebenen) Low-Level-APIs des jeweiligen Datenbank-Herstellers. Bezüglich der Applets verbessert sich die Lage gegenüber der JDBC-ODBC-Bridge dadurch natürlich nicht.
3. **Treiber vom Typ 3** sind rein in Java geschrieben und kommu-

nizieren über ein Datenbank-unabhängiges Netzprotokoll mit einer Middleware-Komponente, die auf dem Server installiert ist. Diese Komponente führt dann den eigentlichen Datenbank-Zugriff durch. Um ein Beispiel zu nennen: Easysoft bietet eine JDBC-ODBC-Bridge vom Typ 3 an. Auf Client-Seite wird ein reiner Java-Treiber verwendet, der mit dem UDP-Protokoll (*user datagram protocol*) via TCP/IP mit dem Server spricht. Server-seitig werden nun die Anfragen des JDBC-Treibers in ODBC-Anfragen transformiert und ODBC führt dann den eigentlichen Datenzugriff weiter.

4. **Treiber vom Typ 4** sind rein in Java geschrieben und kommunizieren mit dem Datenbank-Server über *sein* Netzprotokoll. D.h. der Java-JDBC-Treiber tritt an die Stelle des Datenbank-Clients und wickelt die Kommunikation mit dem Server direkt ab. Ein Beispiel für einen Treiber dieses Typs ist jConnect von Sybase. Der JDBC-Treiber jConnect verwendet das Protokoll TDS (*tabular data stream*) von Sybase und richtet so seine Anfragen direkt an den Datenbank-Server. Treiber vom Typ 4 versprechen die beste Performanz.
   In unserem Beispielprogramm oben verwenden wir den Treiber vom Typ 4 von PostgreSQL.

Connection

Die Klasse `DriverManager` wird verwendet, um eine Connection zu einer Datenquelle herzustellen. Typischerweise verwendet man die Methode

```
DriverManager.getConnection( url, user, passwd )
```

deren Parameter die URL der Datenquelle und die Kennung des Anwenders und sein Kennwort (für die Datenquelle) sind.

Alternativ kann man eine Verbindung zu einer Datenquelle über das Interface `DataSource` herstellen, siehe unten.

URL einer Datenquelle

Der Parameter `url` der Methode `getConnection` bezeichnet einen *Uniform Resource Locator* für eine Datenquelle. Syntaktisch ist der String aus drei Bestandteilen aufgebaut:

```
jdbc:<subprotocol>:<subname>
```

Dabei bezeichnet `<subprotocol>` den Mechanismus, wie die Verbindung zur Datenbank hergestellt wird und `<subname>` beinhaltet die Informationen, die dieser Mechanismus benötigt, um die Verbindung herzustellen. In der Regel ist also das `<subprotocol>` die einem JDBC-Treiber entsprechende Verbindungstechnik und `<subname>` die Bezeichnung der Datenquelle aus Sicht des Treibers.

Ein Treiber kann eine von dieser Konvention abweichende Syntax für den URL der Datenquelle verwenden.

Beispiele für URLs

- Für Sybase jConnect
  `jdbc:sybase:Tds:<Host>:<Port>?ServiceName=<DBName>`.
- Für PostgreSQL
  `jdbc:postgresql:<Database>`
  `jdbc:postgresql://<Host>/<Database>`
  `jdbc:postgresql://<Host>:<Port>/<Database>`

Das Interface `DataSource`

Seit Version 2.0 von JDBC empfiehlt Sun, die Connection zu einer Datenquelle nicht mehr via der Klasse `DriverManager` herzustellen, sondern ein `DataSource`-Objekt zu verwenden. Diese Technik hat folgende Vorteile:

1. `DataSource`-Objekte verwenden einen logischen Namen für die Datenquelle, der durch *JNDI*, den *Java Naming and Directory Interface* auf die eigentliche Datenquelle abgebildet wird.
2. `DataSource`-Objekte bieten Unterstützung für verteilte Transaktionen; die Erweiterung `XADataSource` erzeugt Connections, die an verteilten Transaktionen beteiligt sein können.

Statement

Objekte des Typs `Statement` werden von der `Connection` erzeugt, zu der sie gehören (sollen).

Es kann gleichzeitig mehrere `Statements` innerhalb einer `Connection` geben (jedenfalls bei den meisten JDBC-Treibern).

Abfragende und modifizierende Statements

Die wichtigsten Methoden des Interfaces `Statement` sind:

- `executeQuery` richtet eine Select-Anweisung an das DBMS und erzeugt als Ergebnis ein Objekt vom Typ `ResultSet`
- `executeUpdate` wird für Insert-, Update- und Delete-Anweisungen verwendet. Als Ergebnis gibt die Methode die Anzahl der durch die Anweisung veränderten Zeilen zurück. Man kann diese Methode auch für Anweisungen der DDL (*Data Definition Language*) wie `create table...` verwenden.
- `execute` für anfragende und modifizierende Anweisungen. Der Returnwert ist vom Typ boolean und gibt an, ob als Ergebnis eine Ergebnis-

menge anliegt (erreichbar mit `getResultSet`) oder ein Integer-Wert (abfragbar mit `getUpdateCount`).

- `executeBatch` kann man verwenden, um mehrere modifizierende Anweisungen als Gruppe durchzuführen. Man fügt dem Statement mit `addBatch` Anweisungen hinzu, die dann als sogenannte Batch-Gruppe durchgeführt werden.
- `setQueryTimeout` legt fest, wie lange der JDBC-Treiber warten soll, ehe er eine `SQLException` wirft, weil die Abarbeitung der Anweisung noch nicht abgeschlossen ist.
- `setMaxRows` legt fest, wieviele Zeilen eine Ergenismenge maximal enthalten soll. `0` ist der Default-Wert und bedeutet beliebig viele.

Parametrisierte Statements

SQL-Anweisungen die via `Statement` an das DBMS geschickt werden, werden übersetzt und ausgeführt, wenn die Execute-Methode aufgerufen wird. Oft werden jedoch SQL-Anweisungen derselben Struktur immer wieder verwendet, nur die konkreten Werte sind verschieden. Solche parametrisierte SQL-Anweisungen enthalten an Stelle der (austauschbaren) Werte Platzhalter in Form eines Fragezeichens `?`. Zum Beispiel:

```
select author, title from Books where isbn = ?
```

Mit der Methode `prepareStatement` von `Connection` wird eine parametrisierte Anweisung an das DBMS gegeben und dort übersetzt. Für den eigentlichen Aufruf muss nun natürlich der Platzhalter durch den konkreten Wert ersetzt werden. Dazu gibt es die Methoden `setXXX`, wobei `XXX` für einen Datentyp steht. Die Platzhalter in einem PreparedStatement werden durch ihre Position angesprochen, gezählt von links in der Anweisung beginnend bei 1. Beispiel:

```
PreparedStatement pstmt = con.prepareStatement(
    "select author, title from Books where isbn = ?" );
pstmt.setString( 1, "0-201-70928-7" );
pstmt.executeQuery;
...
pstmt.setString( 1, "3-540-44008-9" );
pstmt.executeQuery;
```

Bemerkung: Man vermeidet Sicherheitslücken, die durch die sogenannte *SQL Injection* ausgenutzt werden, indem man konsequent parametrisierte Anweisungen verwendet.

ResultSet

Ein `ResultSet` repräsentiert einen *Cursor* auf die Ergebnismenge einer Anweisung. Es gibt verschiedene Arten solcher Cursor, die sich in den verschiedenen Arten von `ResultSets` widerspiegeln.

Arten von ResultSets

Die Eigenschaften von Ergebnismengen kann man so einteilen:

- Art der Bewegung des Cursors
  - `TYPE_FORWARD_ONLY`: die Ergebnismenge kann nur sequenziell vorwärts durchlaufen werden. (Ob dabei die Daten geliefert werden, wie sie zum Zeitpunkt des Erzeugens der Ergebnismenge waren, also als Schnappschuss, oder ob die Daten gezeigt werden, wie sie im Moment des Zugriffs auf die aktuelle Zeile in der Datenbnk gültig sind, hängt vom jeweiligen DBMS ab.)
  - `TYPE_SCROLL_INSENSITIVE`: die Ergebnismenge kann vorwärts ode rückwärts durchlaufen werden, beliebige Positionierung ist möglich. Die Ergebnismenge zeigt immer die einmal gefundenen Daten unabhängig von Aktionen anderer Transaktionen.
  - `TYPE_SCROLL_SENSITIVE`: die Ergebnismenge ist „scrollbar“, bei mehrfachem Lesen derselben Zeile werden zwischenzeitliche Änderungen anderer Transaktionen sichtbar.

  Default ist `TYPE_FORWARD_ONLY`.
- Lesender oder ändernder Cursor
  - `CONCUR_READ_ONLY`: Die Daten der Ergebnismenge können über das Interface `ResultSet` nur gelesen werden.
  - `CONCUR_UPDATABLE` : Man kann Methoden von `ResultSet` verwenden, mit denen die Daten der Ergebnismenge verändert werden können.

  Default ist `CONCUR_READ_ONLY`.
- Verhalten bei Transaktionsende
  - `HOLD_CURSORS_OVER`: Beim Commit der aktuellen Transaktion bleibt die Ergebnismenge erhalten.
  - `CLOSE_CURSORS_AT_COMMIT`: Beim Commit der aktuellen Transaktion wird der Cursor geschlosse, damit das Objekt vom Typ `ResultSet`.

  Die Default-Einstellung hängt von der jeweiligen Implementierung ab.

Man gibt die gewünschte Einstellung beim Erzeugen des Statements an. Ferner gibt es Set-Methoden von Statement, um die gewünschte Eigenschaft einzustellen. Beispiel:

```
Statement stmt = con.createStatement(
                    ResultSet.TYPE_SCROLL_INSENSITIVE,
                    ResultSet.CONCUR_READ_ONLY,
                    ResultSet.CLOSE_CURSORS_AT_COMMIT );
```

Verwendung von ResultSets

Übersicht über die Methoden von `ResultSet` – welche erlaubt sind, hängt von den eingestellten Eigenschaften ab.

- Bewegungen des Cursors
  - `next` eine Zeile vorwärts, liefert `false` hinter der letzten Zeile.
  - `previous` eine Zeile rückwärts, liefert `false` vor der ersten Zeile.
  - `first`
  - `last`
  - `beforeFirst`
  - `afterLast`
  - `relative( int rows )` rows vor oder zurück, je nach Vorzeichen.
  - `absolute( int r )` an die `r`-te Zeile.
- Lesen von Werten an der aktuellen Cursorposition
  Es gibt für jeden Datentyp zwei Get-Methoden, eine, die den Index[1] der gewünschten Spalte, die andere, die das Label der gewünschten Spalte als Parameter verwendet. (Das Label ist der Bezeichner, der in der Select-Anweisung hinter `as` angegeben wird – oder, falls fehlend, der Name der Spalte.)
  Beispiel:
  ```
  rs.getString( 1 )
  rs.getString( "author" )
  ```
  Die Methode `findColumn( String columnLabel )` ermittelt die Position der angegebenen Spalte.
- Ermitteln von NULL-Werten
  Die Methode `wasNull` kann man verwenden, um festzustellen, ob der zuletzt mit `getXXX` gelesene Wert tatsächlich in der Datenbank ein NULL-Wert war.
- Ändern von Daten der Ergebnismenge
  `ResultSets` mit der Eigenschaft `CONCUR_UPDATABLE` können über das Interface `ResultSet` verändert werden.
  - Update: Man ändert den Inhalt einer Zeile mit `updateXXX` und schreibt ihn mit `updateRow` in die Datenbank.

[1] Die Positionen der Spalten werden in JDBC beginnend bei 1 gezählt (nicht bei 0), weil dies in SQL so üblich ist.

  - Delete: `deleteRow` löscht die aktuelle Zeile und positioniert den Cursor vor die nächste Zeile ioder ans Ende, wenn die gelöschte Zeile die letzte war.
  - Insert: man erzeugt eine virtuelle leere Zeile zur Eingabe mit `moveToInsertRow`, gibt dort die Werte in die Felder ein mit `updateXXX` und schreibt die neue Zeile mit `insertRow` in die Datenbank.

Darüberhinaus sind sogenannte positionierte Änderungen möglich. Ein Beispiel:

```
Statement stmt1 = con.createStatement();
stmt1.setCursorName( "Cursor1" );
ResultSet rs = stmt1.executeQuery(
  "select author, title, isbn from Books for update of author" );
// bewege den Cursor zur Zeile, die geändert werden soll
while ( ... ) {
  rs.next()
}
// ändern
String cursorName = rs.getCursorName();
Statement stmt2 = con.createStatement();
int updateCount = stmt2.executeUpdate( "update Books " +
  "set author = 'Connelly' where current of " + cursorName );
```

Metadaten zu einer Ergebnismenge

Die Methode `getMetaData` von `ResultSet` erzeugt ein Objekt vom Typ `ResultSetMetaData`, das die Metainformationen zur Ergebnismenge enthält. Die wichtigsten Methoden von `ResultSetMetaData` sind:

- `getColumnCount`: Zahl der Spalten der Ergebnismenge.
- `getColumnLabel`: Label der Spalte.
- `getColumnName`: Name der Spalte.
- `getColumnType`: Datentyp (SQL-Typ) der Werte der Spalte.
- `isNullable`: sind in der Spalte NULL-Werte erlaubt?

Transaktionen

Alle Zugriffe einer SQL-Datenbank auf die Daten werden innerhalb einer Transaktion durchgeführt. Das ist natürlich auch so, wenn der Zugriff via JDBC erfolgt.

Verwendung von Transaktionen

JDBC verwendet per Default den sogenannten Auto-Commit-Modus für die Datenbankzugriffe. Das bedeutet, dass jede SQL-Anweisung in einer Transaktion stattfindet, die sofort nach Ende der Anweisung durch ein Commit beendet wird. In diesem Modus muss also im Anwendungsprogramm der Befehl

```
con.commit(); // Connection con
```

nicht angegeben werden, er wird gewissermaßen automatisch nach jeder SQL-Anweisung eingefügt.

Für viele Anwendungen kommt jedoch der Auto-Commit-Modus nicht in Frage, weil mehrere SQL-Anweisungen innerhalb einer Transaktion verwendet werden müssen. In diesem Fall schaltet man den Auto-Commit-Modus aus

```
con.setAutoCommit( false );
```

und beendet Transaktionen in der Anwendung durch `commit` oder `rollback`.

Eine Vorlage für die Verwendung von Transaktionen ist folgendes Programmiermuster:

```
Connection con = DriverManager.getConnection(
                              url, username, password );
boolean autoCommit = con.getAutoCommit();
Statement stmt;
try {
  con.setAutoCommit( false );
  stmt = con.createStatement();
  stmt.execute(...);
  stmt.execute(...);
  stmt.execute(...);
  ...
  con.commit();
} catch(SQLException sqle) {
  con.rollback();
} finally {
  stmt.close();
  con.setAutoCommit( autoCommit );
}
```

Isolationslevel

JDBC unterstützt die Isolationslevel für Transaktionen, so wie sie im SQL-Standard definiert sind (das tatsächliche Verhalten hängt jedoch vom jeweiligen Datenbankmanagementsystem ab).

Im parallelen Zugriff mehrerer Transaktion auf die Daten können Phänomene der wechselseitigen Beeinflussung auftreten. Der SQL-Standard definiert die Isolationslevel, in dem er festlegt, welche dieser Phänomene im jeweiligen Isolationslevel nicht auftreten können.

Die Phänomene sind:

- „Dirty Read“: eine Transaktion kann Daten einer anderen Transaktion lesen, die noch nicht durch ein Commit bestätigt sind.
- „Non-repeatable Read“: eine Transaktion kann beim wiederholten Zugriff auf Daten veränderte Werte lesen, die eine andere Transaktion erzeugt hat.
- „Phantom Row“: eine Transaktion kann bei der Wiederholung einer Abfrage auf neue Zeilen stoßen, die eine andere Transaktion eingefügt hat.

Im Interface `Connection` werden die 4 Isolationslevel definiert, darüber hinaus gibt es noch `TRANSACTION_NONE`

1. `TRANSACTION_NONE` Der JDBC-Treiber unterstützt keine Transaktionen.
2. `TRANSACTION_READ_UNCOMMITTED` In diesem Isolationslevel können „Dirty Read“ und die beiden anderen Phänomene auftreten. Dieses Isolationslevel darf man nur für lesende Transaktionen verwenden.
3. `TRANSACTION_READ_COMMITTED` In diesem Isolationslevel kann das Phänomen „Dirty Read“ nicht mehr auftreten – die Transaktion kann nur Daten anderer Transaktionen lesen, die durch einen Commit bestätigt sind.
4. `TRANSACTION_REPEATABLE_READ` In diesem Isolationslevel garantiert das DBMS, dass weder „Dirty Read“ noch „Non-repeatable Read“ auftreten kann, d.h. bei wiederholtem Zugriff auf einmal gelesene Daten werden immer wieder dieselben Werte gelesen.
5. `TRANSACTION_SERIALIZABLE` In diesem Isolationslevel garantiert das DBMS, dass keines der Phänomene „Dirty Read“, „Non-repeatable read“ und „Phantom Row“ eintreten kann; eine Transaktion in diesem Isolationslevel läuft unbeeinflusst von anderen Transaktionen ab (wenngleich allerdings eine Verklemmung möglich ist).

In JDBC werden die Isolationslevel als Eigenschaft der `Connection` mit der Methode

```
con.setTransactionIsolation( int level )
```

eingestellt.

Das so eingestellte Isolationslevel gilt dann für alle folgenden Transaktionen. Dies bedeutet jedoch *nicht*, dass *alle* Transaktionen innerhalb einer

Connection *dasselbe* Isolationslevel haben müssen. Es kann mit der genannten Methode verändert werden und somit von der Anwendung nach Bedarf für die jeweilige Transaktion eingestellt werden.

Metadaten der Datenquelle und adaptives Programmieren

Die Connection kann ein Objekt vom Typ `DatabaseMetaData` erzeugen, mit dem man viele Informationen über die Datenquelle erhalten kann.

```
DatabaseMetaData dbmd = con.getMetaData();
boolean b = dbmd.supportsFullOuterJoin();
```

Folgende Informationen kann man erfragen:

- Allgemeine Informationen über Treiber und Datenquelle

  ```
  getURL
  getUserName
  getDatabaseProductVersion getDriverMajorVersion getDriverMinorVersion
  getSchemaTerm getCatalogTerm getProcedureTerm
  nullsAreSortedHigh nullsAreSortedLow
  usesLocalFiles usesLocalFilePerTable
  getSQLKeywords
  ```

- Welche Features unterstützt die Datenquelle?

  ```
  supportsAlterTableWithDropColumn
  supportsBatchUpdates
  supportsTableCorrelationNames
  supportsPositionedDelete
  supportsFullOuterJoins
  supportsStoredProcedures
  supportsMixedCaseQuotedIdentifiers
  supportsANSI92EntryLevelSQL
  supportsCoreSQLGrammar
  supportsMultipleTransactions
  getDefaultTransactionIsolation
  ```

- Schranken für Werte

  ```
  getMaxRowSize
  getMaxStatementLength
  getMaxTablesInSelect
  getMaxConnections
  getMaxCharLiteralLength
  getMaxColumnsInTable
  ```

- Zugriff auf den Systemkatalog

  ```
  getSchemas and getCatalogs
  getTables
  getPrimaryKeys
  getProcedures getProcedureColumns
  getUDTs
  ```

Diese Informationen kann man auch für die *adaptive Programmierung* von Datenbankzugriffen verwenden: Wenn man eine Anwendung schreiben möchte, die mit möglichst jeder JDBC-Datenquelle arbeiten kann, wird man nicht gerne nur den kleinsten gemeinsamen Nenner dieser Datenquellen unterstützen wollen, sondern die Fähigkeiten des jeweiligen Datenbanksystems ausnutzen wollen. Dies kann man tun, indem man die Informationen über die Datenquelle im Programm ermittelt und dementsprechend in der jeweiligen Situation reagiert.

Ein Beispiel für das Konzept:

```
if ( dbmd.supportsFullOuterJoin() ) {
  // ... full outer join
} else {
  // ... Notlösung
}
```

Da diese Technik mit einem hohen Aufwand für die Programmierung und das Testen verbunden ist, wird man natürlich zunächst versuchen, sie zu vermeiden. Aber es kann Situationen geben, wo es unvermeidlich ist, spezielle Eigenschaften einer Datenquelle zu nutzen.