

Ferienkurs JDBC



Inhaltsverzeichnis

1	JDBC	3
1.1	Einführung	3
1.2	Treiber zum Datenbankzugriff	4
1.2.1	Treibertypen	5
1.3	Architekturen mit JDBC	6
1.3.1	Two-Tier-Architecture	7
1.3.2	Three-Tier-Architecture	7
1.4	Datenbankzugriff mit JDBC	8
1.4.1	Verbindung zur Datenbank herstellen	8
1.4.2	Absetzen eines SQL-Statements	11
1.4.3	Auswerten der Ergebnisse	12
1.5	Details wichtiger JDBC-Klassen	14
1.5.1	Connection	15
1.5.2	Die Statement-Klassen	15
1.5.3	Metadaten	16
1.6	Erweiterter Datenbankzugriff	19
A	Literatur-/Quellenverzeichnis	22

1 JDBC

1.1 Einführung

Das JDBC-Package von Sun dient dazu, mit Java-Programmen auf Datenbanken unterschiedlichster Hersteller zuzugreifen. Das Akronym JDBC wird von dem meisten Leuten mit **Java DataBase Connectivity** übersetzt, was aber nicht die offizielle Bezeichnung von Sun ist. JDBC ist von Sun als Trademark registriert.

JDBC ist eine Lowlevel-API¹, mit der man auf einfache Weise SQL-Anweisungen für Datenbanken ausführen kann. JDBC ist außerdem eine Spezifikation, doch wenn man von JDBC redet, meinen die meisten Leute die API. Die JDBC-API ist seit Version 2.0 in zwei Java-Pakete unterteilt:

1. `java.sql`

Dieses Paket enthält die sogenannte JDBC 2.x Core API. Dieser Teil der API ist vollkommen kompatibel zu früheren Versionen der JDBC-API, so daß Programme, die für die alte API entwickelt wurden, ohne Probleme immer noch funktionieren. SUN hat hier vor allem Verbesserungen und Erweiterungen, die SQL betreffen, eingebaut.

2. `javax.sql`

Dieses Paket enthält die sogenannte JDBC 2.0 Standard Extension API. Hier werden Klassen definiert, die den bisherigen Funktionsumfang erweitern, wie zum Beispiel Resultsets zum wahlfreien Zugriff auf Ergebnisse von SQL-Abfragen und ähnliches.

Die JDBC-API gehört zu der sogenannten Java-Core-API², das bedeutet, sie ist frei erhältlich und muß nicht extra gekauft werden. Wenn man also von der Firma SUN das Java Development Kit³ herunterlädt, so ist die JDBC-API mit dabei. Um SQL-Kenntnisse kommt man nicht herum, wenn man mit JDBC auf eine Datenbank zugreifen möchte. Seit JDBC V2.x ist es zwar möglich, auf Datenbanken auch auf andere Weise zuzugreifen, doch **SQL-Kenntnisse** sind trotzdem **notwendig**.

Die API versucht den Programmierer möglichst wenig mit technischen Details wie Aufbau einer Verbindung zur Datenbank und ähnlichem zu belasten, indem sie diese Funktionen in Klassen beziehungsweise Methoden kapselt, die der Programmierer auf einfache Weise benutzen kann. Doch vollkommen abnehmen kann die API dem Programmierer diese Arbeit natürlich nicht. SQL-Statements müssen immer noch verhältnismäßig mühsam in Form von Strings zusammengestückt werden, um sie dann an die Datenbank zu senden.

In den meisten Fällen stellt dies jedoch kein Problem dar, da bei größeren Projekten meist ein oder mehrere Datenbankspezialisten sich um den Zugriff auf

¹Als API bezeichnet man eine Anwendungsprogrammierschnittstelle

²Java-Kern-API – dazu zählt alles was man mit dem JDK von Sun's Webseite herunterladen kann

³JDK



die Datenbank kümmern und eine weitere programmspezifische API erstellen, die dann von allen anderen Teammitgliedern benutzt werden kann. Somit ist der Rest des Teams vollkommen von den Details des Datenbankzugriffs isoliert und benutzt nur noch eine einfache Highlevel-Schnittstelle.

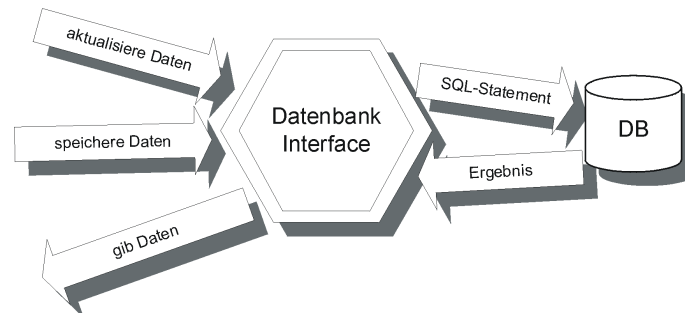


Abbildung 1: Kapselung des DB-Zugriffs

1.2 Treiber zum Datenbankzugriff

Um auf eine Datenbank zugreifen zu können, ist ein sogenannter Treiber notwendig. Die Aufgabe des Treibers ist es, die JDBC-Aufrufe in Aufrufe umzusetzen, die von der jeweiligen Datenbank verstanden werden. In der umgekehrten Richtung muß der Treiber dann auch die Ergebnisse der Datenbankabfragen von der Datenbank entgegen nehmen. Die meisten großen Datenbankhersteller haben inzwischen JDBC-Treiber für ihre Datenbanken. In der JDBC-API sind Interfaces für die Treiber vorbereitet, welche die Datenbankhersteller dann implementieren müssen. Auf Datenbanken, die keinen JDBC-Treiber zur Verfügung stellen, kann man mit Hilfe einer JDBC-ODBC-Bridge zugreifen. Der JDBC-Treiber ist normalerweise immer vom Datenbankhersteller abhängig. Die einzige Ausnahme ist die oben genannte JDBC-ODBC-Bridge, welche die Java-Datenbankabfragen in ODBC-Aufrufe umsetzt. Heutzutage bietet eigentlich jede Datenbank einen ODBC-Treiber an. ODBC ist eine Schnittstelle zur Datenbankprogrammierung für die Programmiersprache C. Treiber von dritten Anbieter können aber durchaus Datenbanken von mehreren Herstellern unterstützen.

Die JDBC-Treiber sind je nach Typ ganz oder teilweise in Java geschrieben und werden vom Programm dynamisch zur Laufzeit geladen. Es ist vollkommen gleichgültig, ob der Datenbankzugriff über den Treiber aus einer Java-Applikation oder einem Applet heraus erfolgt. Es sollte ebenso gleichgültig sein, auf was für eine Datenbank zugegriffen wird. Doch hier gibt es in der Praxis noch einige Probleme:

1. SQL ist zwar standardisiert, doch die unterschiedlichen Datenbanken haben doch noch teilweise unterschiedliche SQL-Dialekte und somit kann man trotz der fest definierten Treiberschnittstelle die Datenbank doch nicht so einfach austauschen.

2. Nicht alle Methoden, die in den Treiber-Interfaces spezifiziert sind, werden von allen Datenbankherstellern implementiert. So kann es passieren, daß man eine Funktion in einem Treiber des einen Herstellers benutzt und diese Methode in dem Treiber eines anderen Herstellers gar nicht implementiert ist.

Von diesen Problemen abgesehen ist es aber möglich, die Datenbank – und somit auch den JDBC-Treiber – einfach auszutauschen, ohne daß davon die Programmierung der Applikation betroffen wäre.

1.2.1 Treibertypen

Es existieren vier verschiedene Typen von Treibern. Bei der Wahl des Treibers ist es wichtig, die Arbeitsweise der Treiber genau zu kennen. Die Schnittstelle des Treibers in Richtung der Anwendung ist die JDBC-Schnittstelle.

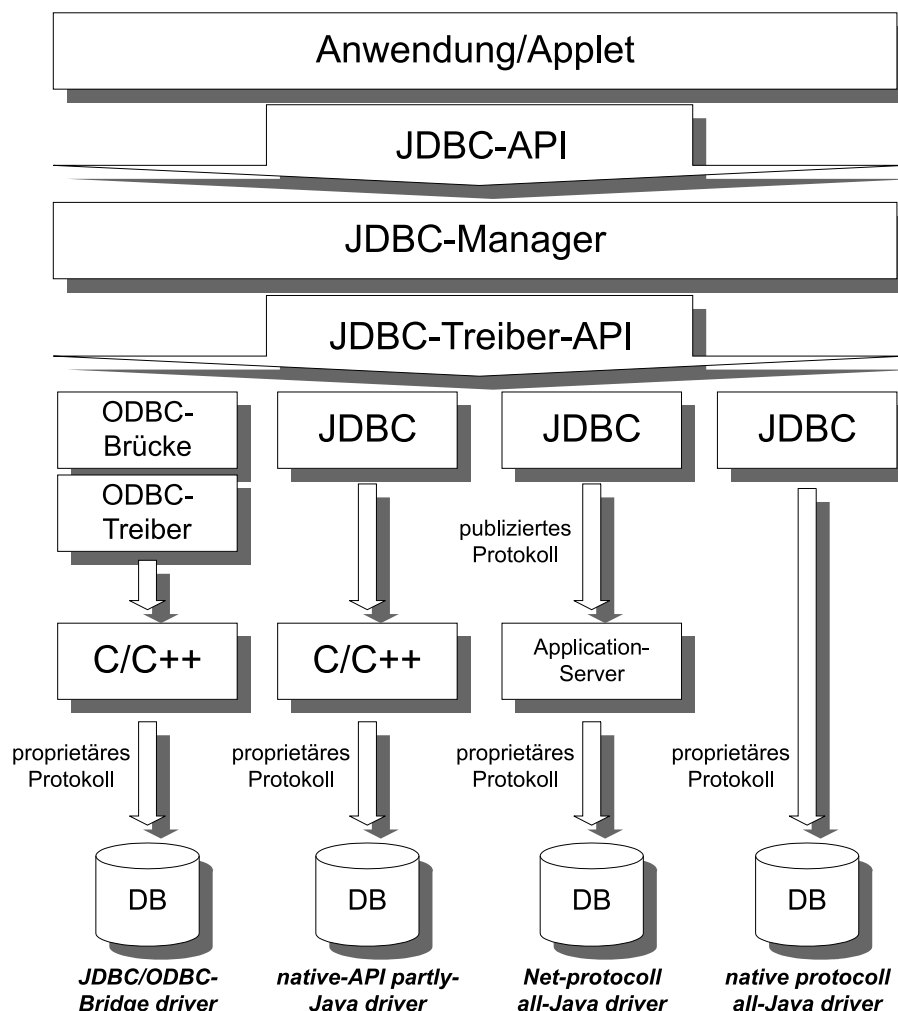


Abbildung 2: JDBC-Treiber

- **Die JDBC-ODBC-Bridge:**

Dieser Treiber stammt von der Firma SUN. Dieser Treiber setzt JDBC-Aufrufe

in ODBC-Aufrufe um. Da man auf fast alle im Handel erhältlichen Datenbanken über eine ODBC-Schnittstelle zugreifen kann, ist mit diesem Treiber der **Zugriff auf fast alle Datenbanken möglich. Dieser Treiber kann nicht direkt über Applets benutzt werden**, doch diese Einschränkung kann man umgehen, indem man das Applet mit einem Application-Server und diesen dann mit der Datenbank verbindet⁴(siehe Abschnitt 1.3).

Dieser Treiber ist aber wohl auch der am wenigsten Performanteste, da hier die JDBC-Aufrufe erst umgesetzt werden müssen. Datenbankzugriffe über JDBC sind nicht gerade bekannt für ihre große Performance, und wenn es in einer Anwendung auf Datendurchsatz ankommt ist die JDBC-ODBC-Bridge sicherlich die schlechteste Lösung.

- **Der Native-API Partly Java-Driver:**
Diese Variante eines JDBC-Treibers **konvertiert JDBC-Aufrufe in API-Aufrufe der entsprechenden Datenbank**. Aufgrund dieses Ansatzes kann dieser Treiber immer nur für eine bestimmte Datenbank funktionieren. Auf dem Client müssen in diesem Fall binäre Treiber installiert werden. Ähnlich wie bei der JDBC-ODBC-Bridge kann auch dieser Treiber **nicht direkt über ein Applet benutzt werden**, sondern nur indirekt über einen Application-Server.
- **Der Net-Protocol All-Java-Driver:**
Bei diesem Ansatz müssen die JDBC-Aufrufe in ein netzunabhängiges Protokoll umgesetzt werden. Dieses netzunabhängige Protokoll wird von einem Application-Server in ein Datenbank-spezifisches Protokoll umgesetzt. Wie der Name schon sagt, ist dieser Treiber **vollständig in Java geschrieben**, also "Pure Java" oder "All Java". Da diese Treibervariante auch **internetfähig** sein soll, müssen zahlreiche zusätzliche Forderungen, wie zum Beispiel erhöhte Sicherheitsanforderungen, Verschlüsselung oder die Unterstützung von Firewalls erfüllt sein.
- **Der Native Protocol All-Java-Driver:**
Bei diesem Treibertyp **werden die JDBC-Aufrufe in ein Netzwerkprotokoll umgesetzt, das direkt von der Datenbank verstanden wird**. Auch dieser Treiber ist ähnlich wie der Net-protocoll All-Java-Driver komplett in Java implementiert und internetfähig, d.h. er kann also auch von Applets aus – ohne den Umweg über einen Application-Server – benutzt werden. Von der Performance her dürfte dieser Treiber wohl am besten sein, doch leider gibt es bisher nur sehr wenige brauchbare Implementierungen dieses Treibertyps.

1.3 Architekturen mit JDBC

Grundsätzlich kann man verteilte Systeme hauptsächlich in zwei verschiedenen Architekturen aufbauen:

1. Two-Tier-Architecture

⁴Three Tier Architektur

2. Three-Tier-Architecture

Dies trifft natürlich auch auf JDBC-Anwendungen zu. Bei diesen Architekturen geht es darum, eine Anwendung in Schichten aufzuteilen. Die erste Schicht ist die, in welcher sich der Benutzer aufhält – der Client, das Frontend oder die GUI. Die zweite Schicht ist die Applikationsschicht. In ihr liegt die eigentliche Logik des Programms. Hier wird richtig gerechnet. Die unterste Schicht ist die sogenannte Datenhaltungsschicht. In unserem Falle ist das die Datenbank, auf die man mit Hilfe von JDBC zugreifen möchte.

Unabhängig davon, welche Architektur man benutzt, sollte – wie schon erwähnt (siehe Seite 3) – ein Programmierer, der die Bedienoberfläche oder die “Logik” einer Anwendung programmiert, eigentlich nichts mit dem Zugriff auf eine Datenbank zu tun haben. Deshalb sollte man die Datenbankaufrufe der Anwendung in einer eigenen API kapseln so, daß die Programmierer nur noch diese Klassen benutzen können.

1.3.1 Two-Tier-Architecture

Die sogenannte Two-Tier-Architektur wird auch 2-Schicht-Modell oder Architektur mit einem Fat-Client genannt. In dieser Architektur werden die Schichten 1 und 2 auf den Client “gelegt” und nur die Datenbank liegt auf dem Datenbankserver. Die Kommunikation zwischen Client und Datenbankserver kann zum Beispiel über JDBC, aber auch über RMI oder Corba stattfinden.

Diese Architektur ist weniger flexibel und skalierbar als die Three-Tier-Architektur. Dafür ist sie aber viel einfacher zu implementieren. Falls diese Architektur für ein Client-Applet benutzt würde, so müßte sehr viel Code vom Server geladen werden, da der JDBC-Treiber, die SQL-Statements und die eigentliche Anwendung benötigt werden.

1.3.2 Three-Tier-Architecture

Diese Architektur wird auch Thin-Client-Architektur genannt, da in diesem Fall beim Client eigentlich nur das Frontend, also die Bedienoberfläche für der User liegt. Die Application wird von einem sogenannten Applikationsserver ausgeführt und die Datenbank wird vom Datenbankserver verwaltet. Die Kommunikation zwischen den Servern und dem Client kann wiederum über die verschiedensten Protokolle ausgeführt werden.

Die Vorteile dieser Architektur liegen vor allem in der Skalierbarkeit. Wenn hier ein paar Clients mehr auf die Applikation zugreifen möchten, stellt das kein Problem dar. Ein weiterer Vorteil ist, da für die Treiber oft Lizenzgebühren pro Installation verlangt werden, daß hier nur der Applikationsserver über einen JDBC-Treiber verfügen muß. Beim Two-Tier-Modell muß bei jedem Client ein Treiber vorhanden sein, das kann bei vielen Benutzern sehr teuer werden, wenn man für jeden Treiber Lizenzgebühren bezahlen muß. Für die gewonnene Flexibilität muß man



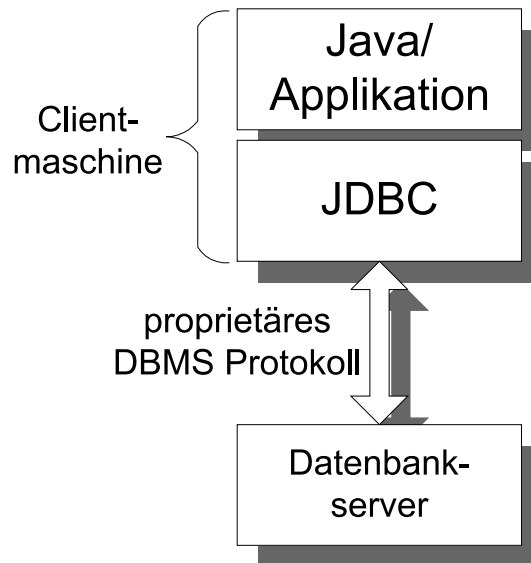


Abbildung 3: 2-Schicht-Architektur

leider auch eine gewachsene Komplexität in Kauf nehmen. Eine Architektur mit einem Drei-Schichtenmodell ist wesentlich komplizierter zu implementieren als ein Zwei-Schichtenmodell. Abbildung 4 zeigt eine Variante des Drei-Schichtenmodell – der Client kann allerdings auch als Java-Applikation implementiert werden, wobei man allerdings ein wenig an Flexibilität verliert, da bei einem Applet bei einer Änderung immer die aktuellste Version vom Server heruntergeladen wird.

1.4 Datenbankzugriff mit JDBC

Der Zugriff auf eine Datenbank läuft immer nach demselben Schema ab:

1. **Herstellen der Verbindung** zur Datenbank
2. **Absetzen eines SQL-Statements** zum Beispiel zum Auslesen, Einfügen oder Ersetzen von Datensätzen in einer Datenbanktabelle.
3. **Auswerten der Ergebnisse**. Dies kann zum Beispiel mit Hilfe eines sogenannten Resultsets geschehen.

Im Folgenden werden die drei Schritte, Verbindung aufbauen, SQL-Statement absetzen und Ergebnisse auswerten näher erläutert. Dieser Abschnitt soll zunächst einmal einen groben Überblick über diesen Ablauf zeigen. Die Details der einzelnen Klassen und wie diese miteinander interagieren wird in Abschnitt 1.5 genauer erläutert.

1.4.1 Verbindung zur Datenbank herstellen

Aus dem Package `java.sql` werden zunächst die Klassen `DriverManager` und `Driver` benötigt. **Die Klasse** `DriverManager` ist dafür zuständig, einen oder auch mehrere Treiber zu registrieren und zu verwalten. Mit Hilfe dieser Klasse kann

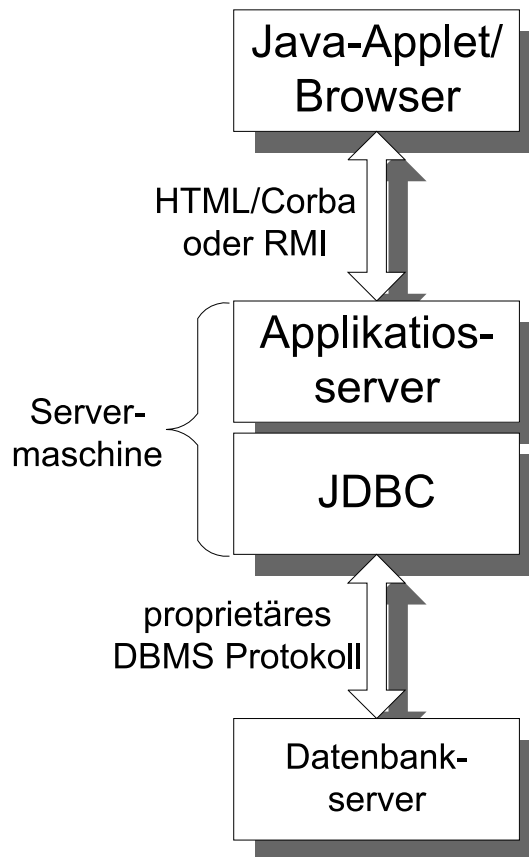


Abbildung 4: 3-Schicht-Architektur

man eine `Connection` erzeugen. **Die Klasse** `Driver` stellt die eigentliche Verbindung zur Datenbank her, nachdem ein `Driver`-Objekt beim `DriverManager` registriert wurde. Jeder Treiber muß in einer Klasse implementiert sein, welche das `Driver`-Interface implementiert. Das Interface garantiert eine immer gleichbleibende Schnittstelle für jeden Treiber. Zum Laden des Treibers muß man lediglich dessen Namen wissen. Jede Datenbank hat einen Treiber mit einem unterschiedlichen Namen. Hier einige Beispiele:

Treibername	DB/Hersteller
<code>sun.jdbc.odbc.JdbcOdbcDriver</code>	JDBC-ODBC-Bridge/Sun
<code>COM.jdbc.odbc.JdbcOdbcDriver</code>	DB2/IBM
<code>postgresql.Driver</code>	Postgres

Tabelle 1: Klassennamen von JDBC-Treibern

Als ersten Schritt beim Aufbauen einer Verbindung zur Datenbank muß man nun den **Datenbanktreiber laden und registrieren**. Um das zu tun, gibt es grundsätzlich zwei verschiedene Möglichkeiten. Die nächste Codezeile zeigt am Beispiel einer DB2-Datenbank, wie man den Treiber lädt und implizit registriert.

VORSICHT: Beim alten JDK V1.1.x ist hier ein Bug und man kann nur die zweite Möglichkeit benutzen.

```
Class.forName('COM.ibm.db2.jdbc.app.DB2Driver');
```

Die zweite Möglichkeit, den Treiber zu laden und beim Treibermanager zu registrieren, ist, ein Objekt der Treiberklasse zu erschaffen und eine Referenz darauf der statischen Methode `registerDriver` des `DriverManagers` zu übergeben.

```
DriverManager.registerDriver(new COM.ibm.db2.jdbc.app.DB2Driver());
```

Der zweite Schritt besteht darin, die **Verbindung zur Datenbank tatsächlich aufzubauen**. Um eine Datenbankverbindung aufzubauen, braucht man als erstes ein `Connection`-Objekt. Dieses Objekt kann vom `DriverManager` mit Hilfe der Methode `getConnection` erzeugt werden. Über dieses `Connection`-Objekt kann dann auf die Datenbank zugegriffen werden.

Die Methode `getConnection` ist statisch und hat drei Parameter. Sie gibt eine Referenz auf ein `Connection`-Objekt zurück. Die drei Übergabeparameter der Methode `getConnection`, sind folgende:

1. Die JDBC-URL der Datenbank – hiermit kann das `Connection`-Objekt die Datenbank finden. Durch diese URL wird der Rechner spezifiziert, auf dem die Datenbank installiert ist.
2. Die User-ID für die Datenbank – Sie wird benötigt, um den Benutzer zu authentifizieren, also festzustellen, ob er derjenige ist, für den er sich ausgibt.
3. Das Passwort für den entsprechenden User.

Die JDBC-URL ist wiederum aus drei Teilen mit folgender Syntax aufgebaut:

```
<Protokoll>:<Subprotokoll>:<Subname>
```

Protokoll, Subprotokoll und Subname sind jeweils durch Doppelpunkte voneinander getrennt. Die JDBC-URL wird benötigt, um die Datenquelle – also die Datenbank – aufzufinden und um festzustellen, welcher konkrete Treiber zum Aufbau der Verbindung benutzt wird. Das Protokoll ist immer `jdbc`. Anhand des Subprotokolls erkennt der `DriverManager`, welchen Datenbanktreiber er zur Verbindungsaufnahme benutzen soll. Der Subname kann ein beliebiges Format haben, welches vom verwendeten Subprotokoll abhängt. Der Subname gibt im allgemeinen den Namen des Rechners an, auf dem die Datenbank installiert ist. Wenn eine Datenbank auf dem lokalen Rechner gemeint ist, so besteht der Subname nur aus der Namen der Datenbank.

Wenn man einen Treiber verwendet, der über das Internet mit seiner Datenquelle kommuniziert, so kann der Subname auch aus einer Internet-URL bestehen. Aber eine JDBC-URL könnte auch so aussehen: `jdbc:db2:sample`. Dies ist ein Beispiel für die URL, falls sich die Datenbank auf dem lokalen Rechner befindet. Dies ist äquivalent zu `jdbc:db2://localhost:6789/sample`. Als Internet-URL, also wenn sich die Datenbank auf einem anderen über das Netzwerk zu erreichenden Rechner befindet, würde sie so aussehen:



jdbc:db2://server:6789/sample.
Der Rechnername ist in diesem Fall server.

Der Code, um eine Datenbankverbindung aufzubauen, könnte zum Beispiel so aussehen:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

String url = "jdbc:odbc:MusterQuelle";
String username = "Mustermann";
String password = "topsecret";

Connection con = DriverManager.getConnection(url, username, password);
```

1.4.2 Absetzen eines SQL-Statements

Der nächste Schritt besteht darin, ein SQL-Statement abzusetzen. Ein SQL-Statement wird als String "zusammengebaut". Es gibt drei verschiedene Arten von Statements:

1. Statement

Dies ist die Vaterklasse von `PreparedStatement`. Sie ist für einfache Statements gedacht. Ein `Statement`-Objekt bekommt man von einer `Connection`, indem man einfach die Methode `executeQuery` aufruft und ihr einen SQL-String übergibt. Hier können keine Parameter angegeben werden. Beispiel:

```
Statement stmt = con.createStatement();
ResultSet result = stmt.executeQuery('SELECT spalte1, spalte2
                                     FROM Tabelle1');
```

2. PreparedStatement

Dieses Statement ist von der Klasse `Statement` abgeleitet. Es unterscheidet sich in mehreren Punkten von seiner Vaterklasse. Zum einen wird die SQL-Anweisung, die dem Konstruktor des `PreparedStatement` übergeben wird, bei der Datenbank vorcompiliert, d.h. es werden schon verschiedene Optimierungen vorgenommen, um möglichst schnell an das Ergebnis der Datenbankabfrage zu kommen. Die Abfrage wird vorbereitet – prepared. Außerdem können bei dieser Statement-Art auch Platzhalter mit "?" gesetzt werden. Den Platzhaltern kann man mit Hilfe verschiedener `set`-Methoden konkrete Werte zuweisen.

Im folgenden Beispiel wird mit Hilfe der Methode `prepareStatement` des `Connection`-Objektes ein neues Objekt vom Typ `PreparedStatement` angelegt. Mit Hilfe der Methode `setString` werden für die Platzhalter die Werte "1" bzw. "Gehaltskürzung" eingetragen. Die fertige SQL-Anweisung setzt überall dort, wo im Feld mit dem Namen `Gefehlt` eine 4 steht, in der Tabelle mit dem Namen `Tabelle1`, das Feld `Vermerk` auf den Wert "Gehaltskürzung". Statements vom



Typ `PreparedStatement` können auch durchaus mehrfach verwendet werden. So wäre es denkbar, daß einer der Platzhalter in einer `for`-Schleife verändert und dann jedesmal die Abfrage mit `executeQuery` erneut ausgeführt wird. Mit der Methode `executeUpdate` wird das Statement dann ausgeführt.

```
PreparedStatement preparedStmt = con.prepareStatement(
    'UPDATE Tabelle1 SET
    Vermerk = ?
    WHERE Gefehlt = ?');
preparedStmt.setString(1, 'Gehaltskürzung');
preparedStmt.setInt(2,4);
int rowCount = preparedStmt.executeUpdate();
```

3. CallableStatement

Die dritte und letzte Statement-Klasse ist von `PreparedStatement` abgeleitet und dazu gedacht, sogenannte Prozeduren auszuführen. Prozeduren sind Datenbankabfragen, die schon kompiliert bei der Datenbank vorliegen und die man nur noch aufzurufen braucht. Parameter werden wiederum mit “?” als Platzhalter übergeben. Zusätzlich muß man noch spezifizieren, ob es sich um reine Eingabe-, Ausgabe- oder Ein-/Ausgabe-Parameter handelt (IN, OUT, IN-OUT).

```
CallableStatement callableStmt = con.prepareCall(
    "call getTestData(?, ?)");
callableStmt.registerOutParameter(1, java.sql.Types.TINYINT);
callableStmt.registerOutParameter(2, java.sql.Types.DECIMAL, 3);
callableStmt.executeQuery();
byte x = callableStmt.getBytes(1);
java.math.BigDecimal n = callableStmt.getBigDecimal(2, 3);
```

1.4.3 Auswerten der Ergebnisse

Zum Auswerten der Ergebnisse von SQL-Abfragen spielt die Klasse `ResultSet` eine zentrale Rolle. Sie enthält die Ergebnisse einer `SELECT`-Abfrage in Tabellenform.

`INSERT`- oder `UPDATE`-Statements liefern nur einen Integer-Wert zurück, der angibt, wie viele Datensätze verändert wurden (Details siehe Abschnitt 1.5.2).

Da das `ResultSet` wie eine Tabelle aufgebaut ist, kann man beim Auswerten der Daten Zeile für Zeile abarbeiten. Dazu muß man aber die Datentypen der einzelnen Felder einer solchen Zeile kennen. Wenn man diese Information nicht hat, kann man sie mit Hilfe der Klasse `ResultSetMetaData` abfragen. Die Zeilen des jeweiligen `ResultSet`s kann man mit Hilfe der Methode `next` erhalten. Auf die einzelnen Felder einer Zeile kann man mit verschiedenen `get`-Methoden zugreifen – für jeden Datentyp gibt es eine `get`-Methode, also zum Beispiel für Integer



`getInt`, oder für Strings `getString`. Welche Spalte einer Zeile gemeint ist, wird der `get`-Methode mittels eines Übergabeparameters mitgeteilt. `getInt(1)` liest also die erste Spalte in der aktuellen Zeile des Resultsets aus und versucht den Rückgabewert in Integer zu konvertieren.

Im folgenden Beispiel sieht man das Zusammenspiel von Objekten der Klassen `Connection`, `Statement` und vor allem `ResultSet` und `ResultSetMetaData`. Der Verbindungsaufbau wurde hier der Übersichtlichkeit halber weggelassen.

Zunächst wird mit Hilfe der Methode `createStatement` des `Connection`-Objekts ein neues `Statement` erschaffen. Dann wird mit `executeQuery` ein SQL-Statement abgesetzt. Das Statement "SELECT * FROM tabelle1" liefert den gesamten Inhalt der Tabelle mit dem Namen `tabelle1` zurück. Mit der Methode `getMetaData` von `ResultSet` bekommt man die Metadaten (siehe Abschnitt 1.5.3), d.h. Spaltennamen und Datentypen der Spalten.

In der `while`-Schleife wird das `ResultSet` Zeile für Zeile abgearbeitet. Die Methode `next` setzt die aktuelle Zeile des `ResultSets` beim ersten Aufruf auf die erste Zeile und bei jedem weiteren Aufruf auf die nächste. Wenn das `ResultSet` komplett abgearbeitet ist und keine weiteren Zeilen mehr vorhanden sind, gibt `next` `false` zurück.

Die Methode `getColumnName` der Klasse `ResultSetMetaData` liefert den Namen der Spalte als String zurück. Die Methode `getColumnType` liefert den Datentyp der angegebenen Spalte als Integer zurück. Die Metadaten und die SQL-Datentypen werden genauer in Abschnitt 1.5.3 beschrieben.

Nachdem man festgestellt hat, ob es sich bei Spalte 1 der Tabelle um ein Stringfeld – SQL-Typ `VARCHAR` handelt, kann man das Feld mit Hilfe der Methode `getString` auslesen.

```
try {
    ...
    String zellenInhalt, spaltenName;
    Integer sqlTyp;

    Statement stmt = connection.createStatement();
    ResultSet result = stmt.executeQuery(SELECT * FROM tabelle1);
    ResultSetMetaData meta = result.getMetaData();

    while (result.next()) {
        // Auslesen des 1. Spaltennamens
        spaltenName = meta.getColumnName(1);
        // Auslesen des SQL-Typs der 1. Spalte
        sqlTyp = meta.getColmnType(1);
        if (sqlTyp == meta.VARCHAR) {
            // Auslesen der ersten Tabellenspalte
```

```

        zellenInhalt = result.getString(1);
    }
    ...
}
stmt.close();
}
catch(SQLException e) {
    // Fehlerbehandlung
}

```

1.5 Details wichtiger JDBC-Klassen

In Abbildung 5 sieht man die Zusammenarbeit der verschiedenen Objekte. Zum Verbindungsaufbau werden Objekte der Klassen `DriverManager`, `Driver` und `Connection` benötigt. Ein Objekt vom Typ `DriverManager` kann mehrere `Driver`-Objekte verwalten. Mit Hilfe eines `Driver`-Objektes kann man über den `DriverManager` ein `Connection`-Objekt erschaffen.

Mit dieser `Connection` kann man verschiedene `Statements` kreieren, die dazu dienen, SQL-Abfragen an die Datenbank zu schicken.

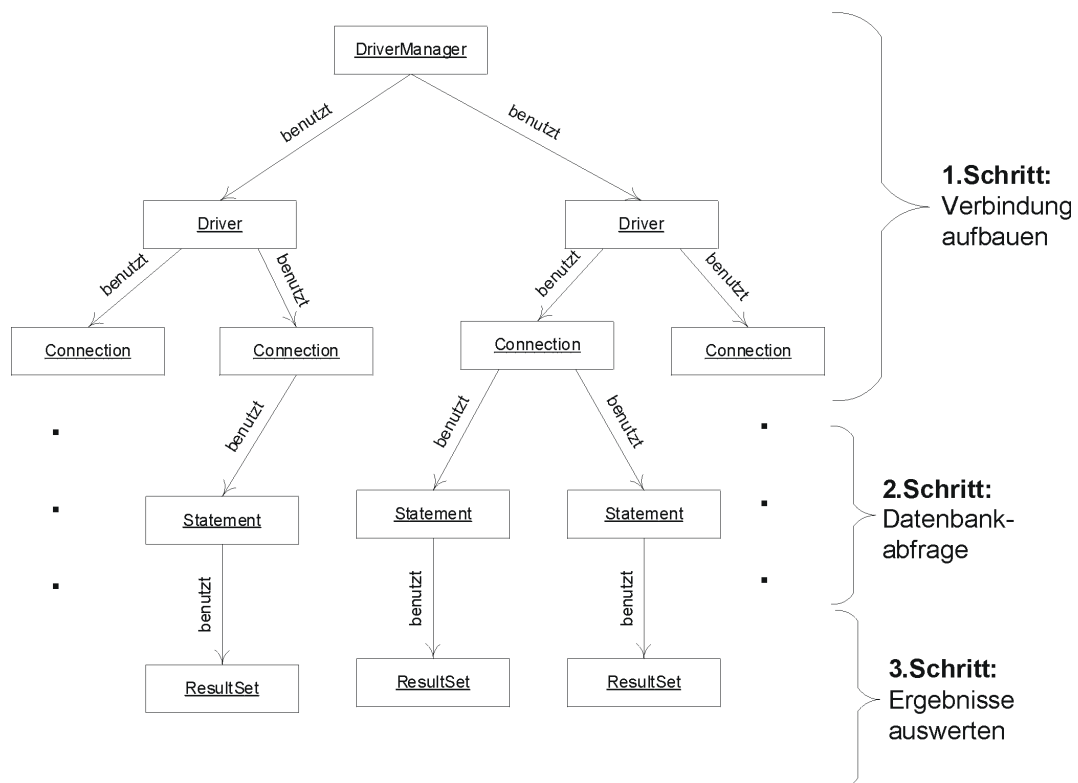


Abbildung 5: Zusammenarbeit wichtiger JDBC-Objekte

1.5.1 Connection

Wie weiter oben schon besprochen (siehe Abschnitt 1.4), ist die Hauptzuständigkeit der Klasse `Connection` die Verbindung zur Datenbank. Eine Applikation kann – wie man aus Abbildung 5 erkennt – mehrere Verbindungen in Form von `Connection`-Objekten zur Datenbank haben.

`Connection`-Objekte sind wiederverwendbar, d.h. wenn einmal eine Verbindung aufgebaut wurde, so kann sie immer wieder verwendet werden, um neue `Statement`-Objekte anzulegen. Die `Connection`-Klasse ist als sogenannte Fabrik [1] implementiert, das heißt, sie erstellt auf Anfrage – Methodenaufruf von `createXXXXXXStatement` – ein `Statement`-Objekt. Da der Verbindungsaufbau zu einer Datenbank über JDBC unter Umständen relativ lange dauern kann (über eine JDBC-ODBC-Bridge mehrere Sekunden) – speichert man aus Performance-Gründen die Referenzen auf `Connection`-Objekte und erzeugt nicht jedesmal, wenn eine neue Transaktion benötigt wird eine neue Verbindung. Dieses Verfahren wird **Connection-Pooling** genannt. Dabei wird eine bestimmte Anzahl Verbindungen zur Datenbank offen gehalten, und wenn ein Client ein SQL-Statement absetzen will, so bekommt er einfach eine Referenz auf eine schon bestehende `Connection` geliefert.

Eine `Connection` stellt einen geschlossenen Transaktionsraum dar. Eine Transaktion ist ein ununterbrechbare Aktion auf einer Datenbank. Wenn man also Datensätze verändert, so wird die Veränderung in Wirklichkeit erst an den Datensätzen vorgenommen, wenn man die Methode `commit` eines `Connection`-Objektes aufgerufen hat. Standardmäßig ist die `Connection` so eingestellt, daß sie nach jedem SQL-Statement ein sogenanntes “autocommit” durchführt, das heißt, sie ruft die `commit`-Methode selbst auf. Dieses Verhalten läßt sich bei der jeweiligen `Connection` einstellen. Wenn während einer Transaktion ein Fehler auftaucht, so kann man sie mit der Methode `rollback` wieder zurücknehmen.

Über das `Connection`-Objekt kann außerdem das Locking-Verhalten der Datenbank beeinflußt werden. Das Locking-Verhalten bestimmt, wie restriktiv sich die Datenbank verhält, wenn zwei Benutzer gleichzeitig auf denselben Datensatz zugreifen.

Über ein `Connection`-Objekt können auch Metadaten über die Datenbank abgefragt werden – aber dazu mehr im Abschnitt 1.5.3.

1.5.2 Die Statement-Klassen

Die Vaterklasse aller `Statement`-Klassen ist die Klasse `Statement`. Sie implementiert die Methoden `executeQuery`, `executeUpdate`, `execute` und seit JDBC 2.0 auch die Methode `executeBatch`. Alle diese Methoden sind zum Ausführen von SQL-Statements auf der Datenbank gedacht. Welche der Methoden man benutzt, hängt ganz von der Art des SQL-Statements ab. **Die Methode** `executeQuery` ist für `SELECT`-Anweisungen, also für Anweisungen, welche ein Ergebnis in Tabellen-



form zurückliefern. In Java wird das Ergebnis einer SELECT-Abfrage, in Form eines Objekts der Klasse `ResultSet` (siehe 1.4.3), repräsentiert. Das SELECT-Statement ist in Datenbanksystemen der wohl am häufigsten gebrauchte Befehl – Benutzer wollen am laufenden Band irgend etwas anschauen, Daten vergleichen Geändert wird an bestehenden Daten verhältnismäßig selten etwas.

Die Methode `executeUpdate` wird für SQL-Statements benutzt, die nur einen Integer-Wert als Rückgabeparameter haben. Also `INSERT`, `UPDATE` und `DELETE`. Der Rückgabewert gibt an, wieviele Datensätze erfolgreich verändert wurden. Aber auch SQL-Befehle wie `DROP TABLE` oder `CREATE TABLE`, zum Löschen bzw. neu Anlegen einer Tabelle können mit `executeUpdate` ausgeführt werden. Der Rückgabewert bei solchen Befehlen ist immer Null. Wenn also keine Exception geworfen wurde, ist der Befehl erfolgreich abgesetzt worden.

Die Methode `execute` ist für gespeicherte Prozeduren vorgesehen – also SQL-Statements, die schon vorkompiliert in der Datenbank vorliegen. Die Methode `execute` liefert eine Array auf `ResultSets` zurück.

Die Methode `executeBatch` ist erst seit JDBC 2.0 vorhanden. Sie dient zum Abarbeiten mehrerer SQL-Statements hintereinander, also in einem Stapel⁵. Auch diese Methode kann dann wie `execute` einen Array von `ResultSets` zurückliefern. Die Methode `executeBatch` ist für die SQL-Statements `INSERT`, `UPDATE` und `DELETE` vorgesehen.

1.5.3 Metadaten

Metadaten sind die Daten über eine Tabelle, also Dinge wie Anzahl der Datensätze, Anzahl der Spalten, Datentyp der Felder, usw. . JDBC hat verschiedene Klassen, um Informationen über die Metadaten einer Datenbank zu erhalten. Mit der Klasse `DatabaseMetaData` lassen sich allgemeine Informationen, die die Datenbank betreffen, auslesen (z.B.: Versionsnummer der Datenbank, Hersteller, maximale Anzahl der zurückgelieferten Zeilen ...). Ein Objekt der Klasse `DatabaseMetaData` kann man über die Methode `getMetaData` der `Connection`-Klasse erhalten.

Das folgende Codebeispiel zeigt, wie man nach dem Aufbau einer Verbindung von der Datenbank Informationen über den Hersteller und den Treiber bekommen kann. Um die Metadaten einer `Connection` zu bekommen, muß man die Methode `getMetaData` der `Connection`-Klasse benutzen.

Mit den Methoden `getDatabaseProductName` und `getDatabaseProductVersion` kann man Produktnamen und Version der aktuellen Datenbank als String bekommen. Informationen über den Treibernamen und dessen Version kann man mit Hilfe der Methoden `getDriverName` und `getDriverVersion` erhalten.

⁵engl. Batch

```
DatabaseMetaData dbMeta = connection.getMetaData();

// Produktname und Produktversion
String produktName = dbMeta.getDatabaseProductName();
String version = dbMeta.getDatabaseProductVersion();
System.out.println("Database Product Name ==> " + produktName);
System.out.println("Database Product Version ==> " + version);

// Treibername und Treiberversion
String treiberName = dbMeta.getDriverName();
String treiberVersion = dbMeta.getDriverVersion();
System.out.println("Driver ==> " + treiberName );
System.out.println("Driver Version ==> " + treiberVersion);
```

Die zweite Klasse für Metadaten bezieht sich auf `ResultSet`, dies ist die Klasse `ResultSetMetaData`. Eine Referenz auf ein Objekt dieser Klasse kann man mit der Methode `getMetaData` des `ResultSet`s erhalten.

Mit den Methoden `getColumnName` und `getColumnType` des `ResultSet`s lassen sich der Name und der Typ der Spalten bzw. Felder eines `ResultSet`-Objektes feststellen. Ein kleines Beispiel hierzu wurde schon in Abschnitt 1.4.3 vorgestellt. Nachdem man festgestellt hat, um welchen Datentyp es sich bei einer bestimmten Spalte handelt, kann man die Informationen der Spalte mit einer `get`-Methode, die dem Datentyp entspricht, auslesen. Welche Methoden für welchen Datentyp geeignet sind, ist in Tabelle 2 dargestellt.

Das folgende Codebeispiel soll nochmals den Gebrauch der Metadaten in Bezug auf die Klasse `ResultSet` zeigen.

Nachdem man über eine `Connection` ein `Statement` erstellt und eine `SELECT`-Abfrage abgesetzt hat, bekommt man eine Referenz auf ein `ResultSet`-Objekt zurück. Dieses Objekt kann man nun mit Hilfe der Methode `getMetaData` nach den Metadaten fragen.

`getColumnName(1)` und `getColumnType(1)` liefern Name und Datentyp der ersten Spalte. Alle Datentypen werden als Integer-Konstanten behandelt. Diese Konstanten sind in der Klasse `ResultSetMetaData` gespeichert. An dem `if`-Konstrukt kann man sehen, wie auf diese Konstanten zugegriffen werden kann.

Die Daten werden dann letztendlich mit der entsprechenden `get`-Methode ausgelesen – hier mit der Methode `getString`.

```
ResultSetMetaData meta = result.getMetaData();

// Auslesen des 1. Spaltennamens
String spaltenName = meta.getColumnName(1);
```



```
// Auslesen des SQL-Typs der 1. Spalte
String sqlTyp = meta.getColumnType(1);

// Auslesen der Daten aus der 1. Spalte
if (sqlTyp == meta.VARCHAR) {
    zellenInhalt = result.getString(1);
}
```

Die folgende Tabelle stellt dar, welche SQL-Datentypen mit welcher Methode eines ResultSet-Objektes ausgelesen werden können. Die "beste" Konvertierung erhält man bei den Methoden, die mit einem fetten **X** gekennzeichnet sind. Alle Paare, die mit **x** gekennzeichnet sind, stellen Möglichkeiten dar, bei denen eine automatische Typkonvertierung erfolgt.

Wie man sieht, kann man mit den Methoden `getString` und `getObject` alle SQL-Datentypen auslesen.

	TinyInt	SmallInt	Integer	BigInt	Real	Float	Double	Decimal	Numeric	Bit	Char	Varchar	Longvarchar	Binary	Varbinary	Longvarbinary	Date	Time	Timestamp
getBytes	X	x	x	x	x	x	x	x	x	x	x	x							
getShort	x	X	x	x	x	x	x	x	x	x	x	x							
getInt	x	x	X	x	x	x	x	x	x	x	x	x							
getLong	x	x	x	X	x	x	x	x	x	x	x	x							
getFloat	x	x	x	x	X	x	x	x	x	x	x	x							
getDouble	x	x	x	x	x	X	X	x	x	x	x	x							
getBigDecimal	x	x	x	x	x	x	x	X	X	x	x	x							
getBoolean	x	x	x	x	x	x	x	x	X	x	x	x							
getString	x	x	x	x	x	x	x	x	x	X	X	x	x	x	x	x	x	x	x
getBytes													X	X	x				
getDate											x	x	x				X		x
getTime											x	x	x					X	x
getTimestamp											x	x	x				x		X
getAsciiStream											x	x	X	x	x	x			
getUnicodeStream											x	x	X	x	x	x			
getBinaryStream													x	x	X				
getObject	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

Tabelle 2: Zulässige Typkonvertierungen mit Hilfe eines ResultSet-Objektes



SQL-Datentyp	Java-Datentyp
CHAR	String
VARCHAR	String
LONGVARCHAR	String
NUMERIC	java.math.BigDecimal
DECIMAL	java.math.BigDecimal
BIT	boolean
TINYINT	byte
SMALLINT	short
INTEGER	int
BIGINT	long
REAL	float
FLOAT	double
DOUBLE	double
BINARY	byte[]
VARBINARY	byte[]
LONGVARBINARY	byte[]
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp

Tabelle 3: Standard-Mapping von SQL-Typen auf Java-Typen

Die Tabelle 4 zeigt das Standard-Mapping der neuen SQL3-Datentypen auf die Java-Datentypen. Diese Datentypen sind seit JDBC V2.1 neu hinzugekommen und werden durch die CORE-API zusätzlich zur alten Funktionalität unterstützt.

SQL-Datentyp	Java-Datentyp
ARRAY	java.sql.Array
BLOB	java.sql.Blob
CLOB	java.sql.Clob
REF	java.sql.Ref

Tabelle 4: Standard-Mapping von SQL3-Typen auf Java-Typen

1.6 Erweiterter Datenbankzugriff

Der sogenannte erweiterte Datenbankzugriff bezieht sich auf die Klassen aus dem Paket `javax.sql`. Dieses Paket wird auch "Optional Package" oder früher "Standard Extension" genannt. Dieses Paket muß extra von SUN heruntergeladen werden. Im Folgenden werden hier die verschiedenen Möglichkeiten des Datenbankzugriffs mit diesen Klassen gezeigt. Das JDBC 2.0 Optional Package bietet unter anderem folgende Features:

- Das `DataSource`-Interface um mit dem JNDI (Java Naming and Directory Interface) zu arbeiten, was eine bessere Möglichkeit darstellt, um eine Verbindung



Java-Datentyp	SQL-Datentyp
String	VARCHAR oder LONGVARCHAR
java.math.BigDecimal	NUMERIC
Boolean	BIT
byte	TINYINT
short	SMALLINT
int	INTEGER
long	BIGINT
float	REAL
double	DOUBLE
byte[]	VARBINARY oder LONGVARBINARY
java.sql.Date	DATE
java.sql.Time	TIME
java.sql.Timestamp	TIMESTAMP

Tabelle 5: Standard-Mapping von Java-Typen auf SQL-Typen

zu Datenquellen herzustellen. Der Sinn dieser neuen Methode ist es, daß der Code lesbarer, leichter zu portieren und leichter zu pflegen wird.

- Die `RowSet`-Interface stellt einen Container dar, der einen Satz Zeilen (engl. Rows) enthält. Die `RowSets` stellen praktisch verbesserte `ResultSet`s dar. Ein `RowSet` ist eine Bean-Komponente und ermöglicht wahlfreien Zugriff, also Scrolling vorwärts, rückwärts
- Connection Pooling ist ein Mechanismus bei dem vorhandene Verbindungen zur Datenbank nicht einfach geschlossen werden, sondern bei Bedarf wiederverwendet werden. Eine neue Verbindung zur Datenbank aufzubauen, ist eine sehr zeitaufwendige Aktion, deshalb kann Connection-Pooling die Performance erheblich steigern.
- Verteilte Transaktionen
- (Neue (SQL-3) Datentypen (siehe Tabelle 4)
Ist bei Core-API-dabei.
 - **Array**
 - **Blob**
Ein Blob ist das Java-pendant zu dem gleichnamigen SQL-Datentyp. Ein SQL-BLOB ist ein Datentyp, der sogenannte "Binary Large Objects" als Wert in einer Spalte einer Datenbanktabelle speichert [2]. Mit Hilfe dieses Datentyps könnte man zum Beispiel ausführbare Dateien in einer Datenbank abspeichern oder Multimediadaten wie Audio- oder Video-Dateien.
 - **Clob**
Auch ein Clob stellt das Java-pendant zum gleichnamigen SQL-Datentypen dar. Ein SQL-Clob speichert sogenannte "Character Large Objects" in einer Datenbanktabelle[2].

- Ref
- Struct

A Literatur-/Quellenverzeichnis

Literatur

- [1] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides Entwurfsmuster
- [2] Java(TM) 2 Platform, Standard Edition, v1.2.2 API Specification
- [3] Maydene Fischer JDBC Data Access API – The JDBC 2.0 Optional Package
- [4] www.corba.ch/e/3tier.html
- [5] www.mgt.buffalo.edu/software/Client_Server/cs3tier.htm
- [6] Diplomarbeit Cornelia Weiss FHTE 1999
- [7] Diplomarbeit Tobias Rieck FHTE 2000
- [8] Diplomarbeit Carsten Stein FHTE 2000