In diesem Kursteil

- Einführung in JDBC
  - SQL
  - ODBC
  - Java und JDBC
  - JDBC 1.0
  - JDBC 2.0
- Aufsetzen einer Datenbank
  - ODBC
  - Kreieren einer Datenbank
  - Kreieren einer Tabelle
  - Einfügen von Daten in die Tabelle / DB
  - Abfragen der Datenbank
- Verbindung zur Datenbank aus Java herstellen
- Kreieren von Tabellen
- Lesen von Daten aus dem ResultSet
- Mutieren von Tabellen
- Zusammenfassung der Grundlagen
- Einsatz von PreparedStatements
- Joins
- Transaktionen
- Stored Procedures
- Wie geht's weiter?

JDBC™ ODBC Grundlagen und Praxis

.

1.1. Kursübersicht

In diesem Modul besprechen wir

- die Hintergründe, die zu JDBC führten.
- die grundlegenden Konzepte zum Thema Datenbanken und JDBC Applikationen.
- wie JDBC Klassen zusammenarbeiten, um mit Datenbanken zu kommunizieren.
- einige fortgeschrittene Konzepte zu JDBC und Datenbanken.
- wie man grundlegende Möglichkeiten von JDBC 2.0 einsetzen kann.

In diesem Kurs verwenden wir ausschliesslich die JDBC-ODBC Bridge, um auf Daten zuzugreifen.

1.1.1. Kursvoraussetzungen

Sie sollten vertraut sein mit den Konzepten der objektorientierten Programmierung im Allgemeinen und der Java Programmiersprache im Speziellen.

Die Beispiele zu JDBC setzen voraus, dass Sie Java Programme verstehen, modifizieren und selber schreiben können. In der Regel benötigen Sie dazu mittlere Java Kenntnisse. So oder so finden Sie jeweils Beispiele auf dem Server / der CD, als Leiter für eigene Beispiele.

Von Vorteil wäre es, wenn Sie bereits Kenntnisse über Relationale Datenbank Managemenet Systeme hätten und die Structured Query Language (SQL) mindestens ansatzweise kennen würden. Im Anhang finden Sie weitere Informationen zu SQL.

Für die Beispiele in diesem Modul setzen wir die JDBC-ODBC Bridge ein, welche mit JDK ausgeliefert wird.

Zusätzlich benötigen Sie
- ODBC
- Excel
- Access

sowie natürlich einige Java Kenntnisse.

1.1.1.1. Lernziele

Nach dem Durcharbeiten dieses Moduls sollten Sie in der Lage sein:
- Datenbank Tabellen zu kreieren und zu modifizieren.
- auf Informationen in einer Datenbank zugereifen können und die darin enthaltenen Daten modifizieren können.
- dynamisch Informationen über eine Datenbank und die darin enthaltenen Daten zu erhalten.
- die Fehlerbehandlung und Ausnahmen von JDBC und deren Einsatz zu kennen.
- prepared Statements einzusetzen.
- Transaktionen zu definieren und mehrere Operatione zu einer Transaktion zusammenzufassen.
- Batch Updates zu programmieren und Scrollable ResultSets einzusetzen.

1.1.1.2. Benötigte Software

Sie benötigen ein aktuelles JDK plus Windows plus
- ODBC
- Excel
- Access

JDBC ist eine der wenigen Java Technologien , bei denen Sie auf externe Implementationen angewiesen sind. Daher werden Sie gelegentlich speziell gekennzeichnete Hinweise auf die Details der einen oder anderen Datenbank finden.

Excel und Access sind Markennamen von Microsoft.

1.1.2. Einführung in JDBC™

JDBC™ ist ein Java™ API (Application Programming Interface), eine Beschreibung eines Standard Frameworks für die Bearbeitung von tabellarischen und allgemeiner, präziser gesagt, relationalen Daten. JDBC 2.0 macht SQL dem Programmierer semi-transparent. SQL ist immer noch die *lingua franca* der Standard Datenbanken und stellt einen grossen Fortschritt dar, auf dem Weg der Trennung der Daten von den Programmen. Bevor wir mit dem eigentlichen Kurs anfangen, schauen wir noch einmal kurz zurück.

1.1.2.1. SQL

SQL ist eine standardisierte Sprache, um relationale Datenbanken zu kreieren, zu manipulieren, abzufragen und zu verwalten. In diesem Modul werden wir nicht sehr tief auf SQL eingehen. Einige Grundlagen werden aber wiederholt, so dass der Modul möglichst selbständig genutzt werden kann.

Folgende Begriffe sollten Sie mindestens grob kennen:

- eine *Datenbank* ist im Wesentlichen ein smarter Container für Tabellen.
- eine *Tabelle* ist ein Container, der aus Datensätzen, "Zeilen", besteht.
- eine *Zeile* ist (konzeptionell) ein Container bestehend aus Spalten.
- eine *Spalte* ist ein einzelnes Datenelement mit Namen, Datentyp und Wert.

Sie sollten sich mit diesen Begriffen vertraut machen und die Unterschiede kennen. Aber am Anfang reichen einfachste Kenntnisse: eine Datenbank entspricht grob einem Dateisystem; eine Tabelle entspricht einer Datei; eine Zeile (row) entspricht grob einem Datensatz; eine Spalte (column) endspricht einem Datenfeld oder einer Variable.

Im Zusammenhang mit diesen Begriffen sollten Sie auch die *Input/Output (I/O) Operationen* von Java kennen.

Weil SQL eine anwendungsspezifische Sprache ist, kann eine einzelne SQL Anweisung zu komplexen Berechnungen und umfangreichen Datenmanipulationen führen, speziell beim Sortieren oder Einfügen neuer Daten. SQL wurde 1992 standardisiert. Damit besteht die prinzipielle Möglichkeit, Programme für mehrere Datenbanken zu entwickeln und ohne Änderungen ausführen zu können. Um SQL einsetzen zu können, muss man aber mit einer SQL Datenbank verbunden sein und daher wird man in der Regel die jeweils speziellen Erweiterungen dieser Datenbank nutzen.

1.1.2.2. ODBC

ODBC (Open Database Connectivity) entspricht in etwa einem C-basierten Interfacezu SQL-basierten Datenbanken und stellt ein konsistentes Interface für die Kommunikation mit Datenbanken und den Zugriff auf deren *Metadaten* zur Verfügung. Die Metadaten beschreiben die Datenbank und deren Tabellen, Datentypen und beispielsweise Indices.

Datenbankhersteller stellen spezifische Treiber oder "Bridges" für verschiedene Datenbanken zur Verfügung. Damit kann man mit Hilfe von SQL und ODBC auf eine standardisierte Art und Weise auf Datenbanken zugreifen. ODBC wurde für PCs entwickelt, ist heute aber ein de facto Industriestandard.

Obschon SQL gut geeignet ist Daten und Datenbanken zu manipulieren, ist SQL keine vollständige Programmiersprache. Die Sprache dient lediglich der Kommunikation mit der Datenbank. Sie benötigen also eine weitere, vollständige Programmiersprache, um die Verbindung mit der Datenbank aufzunehmen und die Ergebnisse aufzubereiten und eventuell zu visualisieren.

Falls Sie plattformunabhängig entwickeln wollen, ist Java eine gute Wahl, speziell für solche Datenbankanwendungen. Falls Sie lediglich eine Plattform benötigen, kann C++ genau so gut geeignet sein.

1.1.2.3. Die Java™ Programmier Sprache und JDBC

Ein Java Programm kann auf unterschiedlichen Plattformen ausgeführt werden. Das gestattet es Ihnen Datenbankprogramme zu schreiben, welche universell einsetzbar sind. Einzig das Package `java.sql` oder JDBC, auch als portable Version von ODBC angesehen, muss vorhanden sein.

Bemerkung 1

Obschon Sie portable Applikationen mit standardisierten Datenbankinterfaces schreiben können, müssen Sie beachten, dass nicht zuletzt aus Konkurrenzgründen, die Datenbanken

selbst, inkompatibel bleiben, selbst auf der Stufe SQL! Sie müssen also versuchen, den kleinsten gemeinsamen Nenner der von Ihnen eingesetzten Datenbanken zu finden. Dieses Problem besteht unabhängig davon, ob Sie OBDC, JDBC und SQL mit Java oder prorietäre Protokolle verwenden.

Bemerkung 2

*Neben JDBC existiert für ORACLE ein sogenanntes Embedded SQL für Java,*SQLJ *SQLJ ist Oracle's Implementation des SQLJ Standard, der die Integration von SQL Statements in Java Programmen definiert. SQLJ empfinden viele als kompater als JDBC Anwendungen. Aber seien Sie gewarnt: neben ORACLE werden Sie kaum einen Hersteller finden, der SQLJ unterstützt. SQLJbesteht aus Java Source Code in den SQLJ eingebettet ist.Der SQLJ PreCompilerübersetzt das SQLJ in äquivalente JDBC Aufrufe.*

Ein *JDBC Driver* ist eine Klasse (bezw. mehrere Klassen, falls man darunter den vollen DBMS Zugriffsmechanismus versteht), welche das JDBC Driver Interface implementiert. Zudem muss der Treiber die JDBC Aufrufe in datenbankspezifische Aufrufe umsetzen. Der Treiber muss also die gesamte Arbeit erledigen. Es sind mehrere Treiber in Java erhältlich, so dass Sie ausgehend davon eigene Treiber schreiben können.

Gemäss JDK existieren vier verschiedene Treiber Typen für JDBC. Diese sind in der aktuellen Beschreibung von JDBC enthalten. Viele Datenbankanbieter stellen heute JDBC Treiber für ihre Datenbanken zur Verfügung. Daneben existieren auch neutrale Treiber, die universell einsetzbar sind und in der Regel auf ODBC abbilden. Einige Hersteller haben sich auch auf die Entwicklung von Datenbanktreibern spezialisiert und bieten diese unabhängig von der Datenbank an.

Die vier JDBC Treibertypen sind:

1. *JDBC-ODBC bridge plus ODBC driver*
2. *Native-API partly-Java driver*
3. *JDBC-Net pure Java driver*
4. *Native-protocol pure Java driver.*

JDBC hat sich über die letzten Jahre signifikant weiter entwickelt. Die erste Version war bereits mit der ersten Version von JDK erhältlich. Die aktuellste Version finden Sie bei Sun: http://java.sun.com/products/jdbc/index.html

1.1.2.4. JDBC 1.0

Das JDBC 1.0 API stellte eine einfache Basis eines Frameworks zur Verfügung, um Daten abzufragen und SQL Anweisungen auszuführen. Dazu wurden Schnittstellen für folgende Funktionen zur Verfügung gestellt:

- Driver
- DriverManager
- Connection
- Statement
- PreparedStatement
- CallableStatement
- ResultSet
- DatabaseMetaData
- ResultSetMetaData
- Types

Wie Sie sehen werden, wird ein Treiber an den `DriverManager` übergeben. Dieser stellt dann eine Verbindung her, liefert also ein `Connection` Objekt. Falls Sie dann ein `Statement`, `PreparedStatement` oder `CallableStatement` erstellen, dann können Sie auf die Datenbank zugreifen und die Daten manipulieren.

Eine Abfrage liefert Ihnen Daten zurück, die Sie eventuell weiter aufbereiten müssen. Durch Zugriff auf die Metadaten und die `ResultSet` Metadaten erhalten Sie detailliertere Informationen über die Datenbank und die selektierten Daten Ihrer Abfrage.

1.1.2.5. JDBC 2.0

Die JDBC 2.0 API Dokumentation besteht aus zwei Teilen: dem *core* API, welches wir hier besprechen und das JDBC 2.0 Optional Package. Im Allgemeinen befasst sich das JDBC 2.0 core API primär mit Performance, Klassenerweiterungen und Funktionalitäten sowie SQL3 (SQL-99) Datentypen.

Neue Funktionalitäten im core API umfassen Scrollable Result Sets, Batch Updates, verbesserte Funktionalitäten zum Einfügen, Löschen und Mutieren der Daten, sowie der Internationalisierung, `java.math.BigDecimal` und verschiedene Zeitzonen.

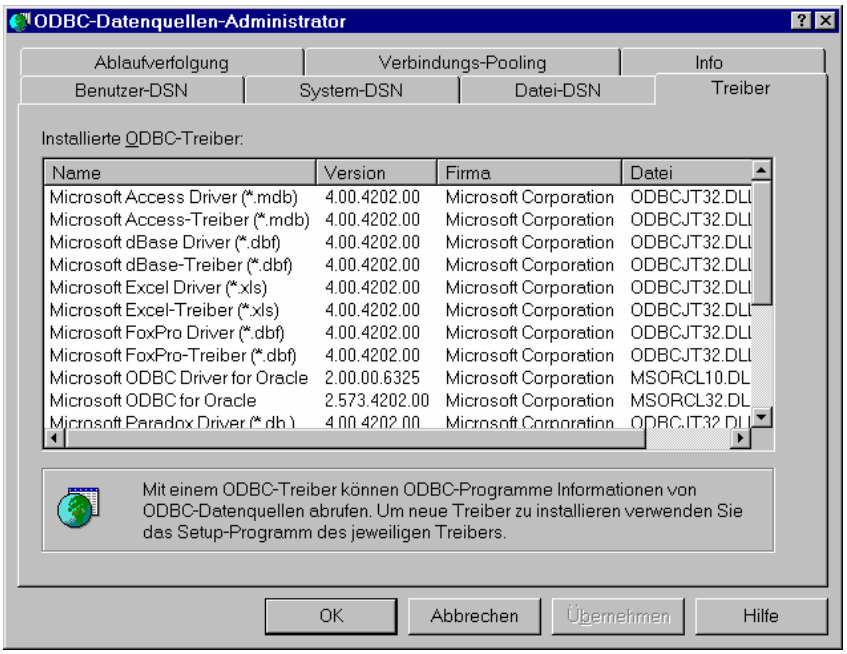
- Das `java.sql package` ist das JDBC 2.0 core API. Es umfasst das ursprüngliche JDBC API, also JDBC 1.0 API, plus neue core APIs, welche später hinzukamen. Diese Teile sind Bestandteil des JDKs..
- Das `javax.sql` package ist das JDBC 2.0 Standard Extension API. Dieses Package ist völlig neu und als separater Download erhältlich oder als Teil von Java 2 Platform SDK, Enterprise Edition.

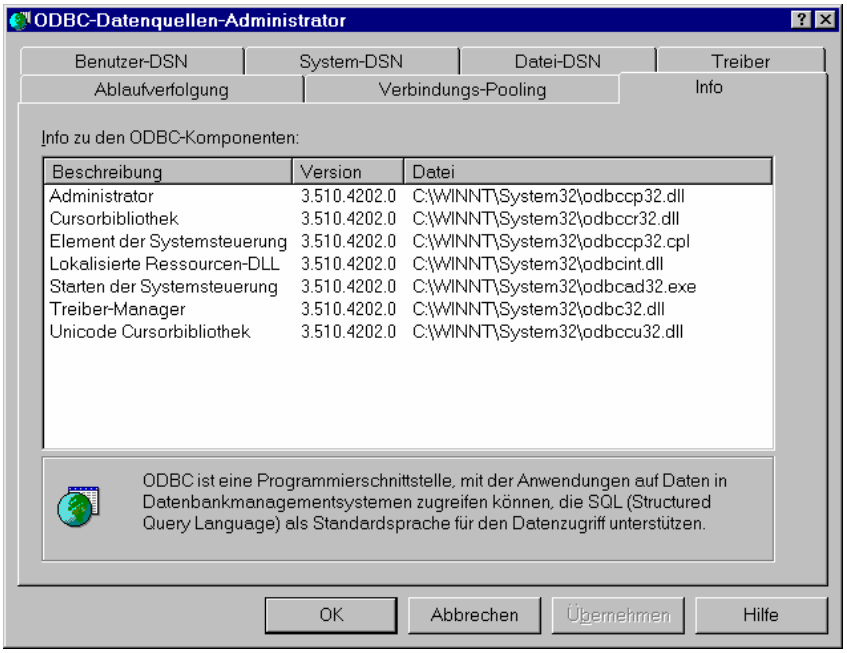
1.2. Aufsetzen einer Datenbank

Als erstes müssen wir eine Datenbank installieren. Dazu wollen wir in diesem Modul einfach alles ODBC fähigen Quellen verwenden. Daher müssen wir als erstes ODBC aufsetzen das heisst installieren.

Sie finden auf dem Server / der CD die aktuelle Version der dlls. Prüfen Sie in der Systemsteuerung, ob die verschiedenen dll's die gleiche Version haben:

und





1.2.1. Einrichten von ODBC-Datenquellen

Um auf die Inhalte einer Datenbank zugreifen zu können, muss eine ODBC-Datenquelle erstellt und konfiguriert werden, die auf die gewünschte Datenbank verweist. Rufen Sie dazu den ODBC-Administrator auf, der sich in der Systemsteuerung von Windows befindet.
Nach dem Start des Administrators finden Sie mehrere Registerkarten vor. Für das Einrichten einer Datenquelle sind die DSN-Register wichtig (DSN = data source name -> Datenquellename).

1.2.1.1. Übersicht

Alle drei Register dienen dem Einrichten, Konfigurieren und Löschen von Datenquellen. Sie unterscheiden sich jedoch in ihrer Zugriffsart.

Benutzer-DSN:
Benutzerdatenquellen sind nur für den Benuzter sichtbar und können nur auf dem aktuellen Computer verwendet werden.

System-DSN:
Auf Systemdatenquellen können alle Benutzer und NT-Dienste zugreifen.

Datei-DSN:
Dateidatenquellen sind auf Dateien basierende Datenquellen, die alle Benutzer, die die gleichen Treiber installiert haben, gemeinsam nutzen können, um somit Zugriff auf die Datenbank zu haben.

1.2.1.2. Hinzufügen einer ODBC-Datenquelle

Um bei Verwendung eines Webservers auf ODBC-Datenquellen zugreifen zu können, wählen Sie die unter NT wie unter Win9x die Registerkarte "System-DSN".
Klicken Sie in der Registerkarte auf die Schaltfläche Hinzufügen (siehe nachfolgende Abb.), um eine neue ODBC-Datenquelle einzurichten. Wie Sie sehen, enthält die Registerkarte bereits eine System-Datenquelle.

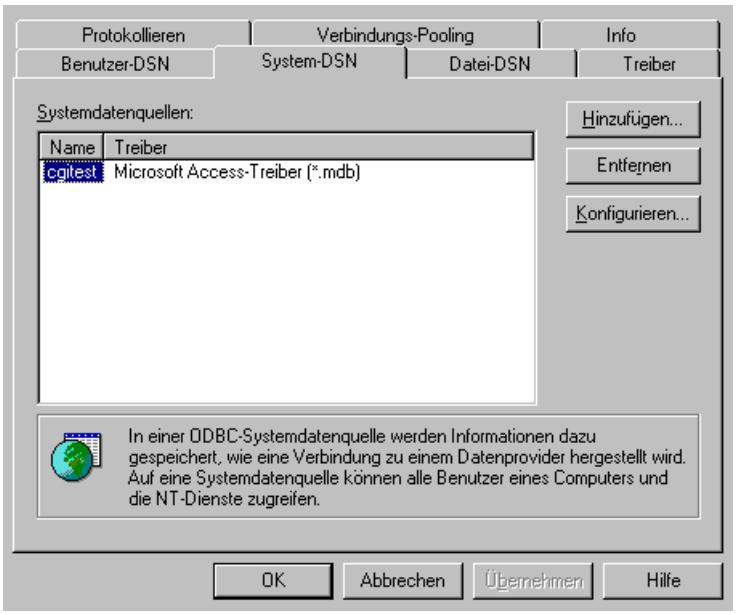


Abbildung 1: Der ODBC-Datenquellen-Administrator

Es öffnet sich ein weiteres Dialogfeld, in dem Sie den Datenbanktreiber für die neue Datenquelle auswählen (siehe Abb. 2).

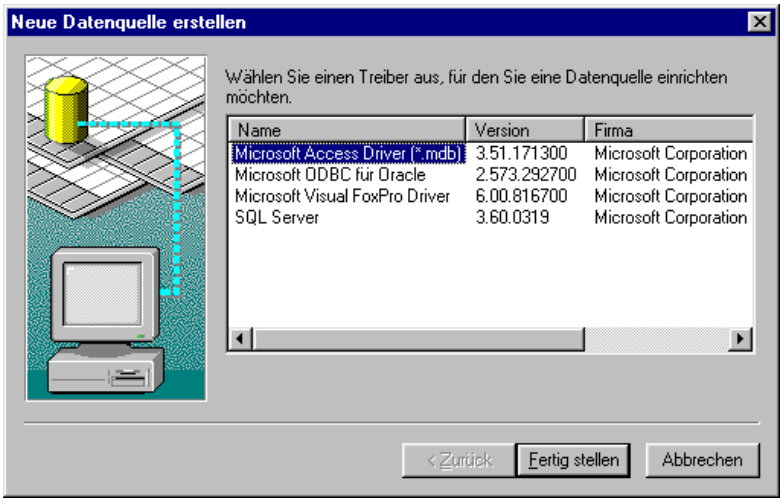


Abbildung 2: Das Dialogfeld Neue Datenquelle erstellen

Möchten Sie z.B. eine Access-Datenbank verwenden, markieren Sie den Microsoft Access-Treiber und klicken auf die Schaltfläche **Fertigstellen**.

Im Dialogfeld ODBC Microsoft Access 97-Setup (siehe Abb. "ODBC_Access_Setup") geben Sie einen kurzen, einfachen Namen für die Datenquelle an. Ihre Anwendung verwendet diesen Namen, um die für die Datenbankverbindung zu verwendende Konfiguration der ODBC-Datenquelle zu spezifizieren. Daher sollte der Name den Zweck der Datenbank umreißen oder auf die Anwendung hinweisen, die mit der Datenbank arbeitet.

Abbildung "ODBC_Access_Setup": Das Dialogfeld ODBC Microsoft Access 97-Setup

Nachdem Sie einen Namen und optional eine Beschreibung für die Datenquelle eingegeben haben, müssen Sie festlegen, wo sich die Datenbank befindet. Klicken Sie auf die Schaltfläche **Auswählen**, und spezifizieren Sie dann die Access-Datenbank, die Sie bereits erstellt haben. Nachdem Sie die ODBC-Datenquelle für Ihre Datenbank konfiguriert haben, klicken Sie auf die Schaltfläche OK, um die neue Datenquelle in den ODBC-Datenquellen-Administrator aufzunehmen. Klicken Sie erneut auf OK, um die Konfiguration zu beenden und den ODBC-Datenquellen-Administrator zu schließen.
Die Datenquelle ist nun eingerichtet.

Analog können Sie die Datenquellen für Textdateien oder Oracle oder ... einrichten. In ODBC werden verschiedene ODBC Treiber bereits vorinstalliert, beispielsweise für Access und Text, Excel und viele mehr. Sie müssen Ihre Installation überprüfen!

1.2.2. Installation von JDBC und ODBC

Die Installation von ODBC und JDBC bedingt bestimmte dll's auf der ODBC Seite und bestimmte Klassen (sql*.*) auf der Java Seite.Diesen Teil werden wir mit einem einfachen Verbindungsaufbau überprüfen. Voraussetzung ist, dass das JDK korrekt installiert wurde. Wir möchten auf drei unterschiedliche Datenquellen zugreifen können:

- Auf ein Access File mit dem Namen `SQLIntro`. Diese Datenbank enthält nur eine Tabelle "Mitarbeiter" und einige wenige Datensätze. Dadurch haben Sie die Möglichkeit den Zugriff zu testen und auch beliebige Erweiterungen vor zu nehmen. Später ergänzen wir die Access Datenbanken durch eine COFFEEBREAK (oder COFFEESHOP) Access Datenbank.

- Auf ein Excel Spreadsheet. Dieses Beispiel ist die (aktuelle) Klassenliste. Der Zugriff erfordert einiges an Änderungen : es muss ein Namen für den gewünschten Datenbereich eingefügt werden. Sonst funktioniert der Zugriff nicht. Details sind auf dem Übungsblatt.

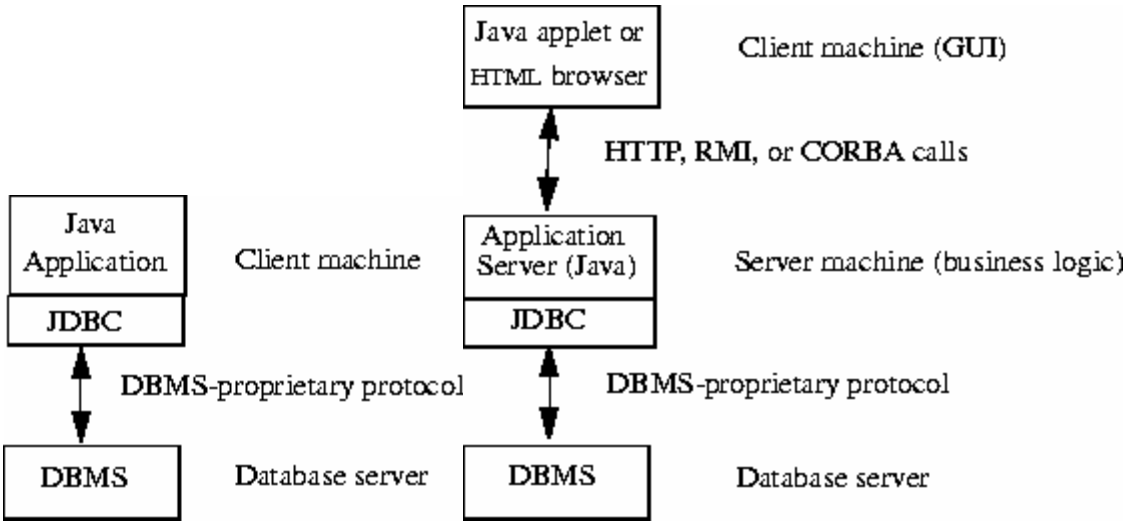
- auf eine Textdatei, die als ODBC Datenquelle definiert wurde und den Text im CSV Format speichert ("Semikolon getrennte Daten").

Das Aufsetzen von ODBC und JDBC und der Bridge ist nicht trivial. Aber da die Software schichtweise benutzt wird, ist es auch möglich sie schichtweise zu testen. Wie dies geschehen kann, wird in der Übung besprochen und beschrieben.

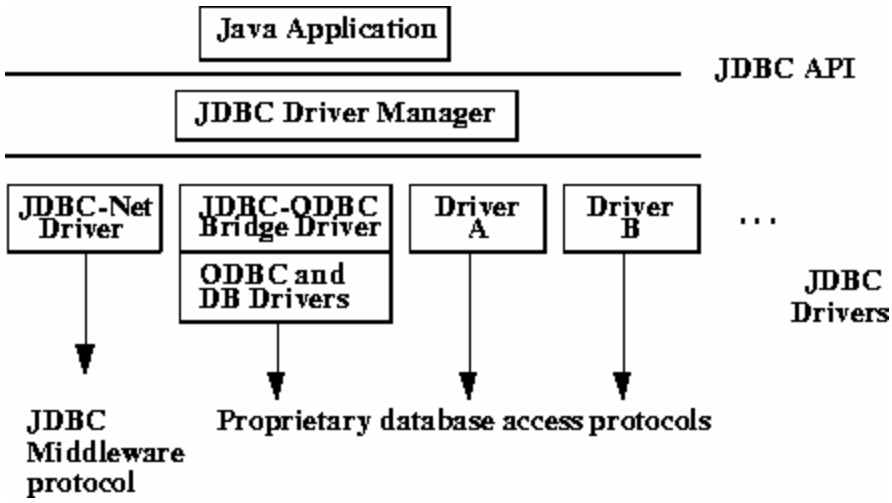
1.2.2.1. ODBC versus JDBC

Der Datenbankzugriff mit Hilfe von ODBC ist in Java eher schlecht möglich, weil die call Conventionen in ODBC sich an C orientierten und damit für Java ungeeignet sind. JDBC stellt also quasi eine Bridge zur Datenbank dar, die auch durch ODBC funktioniert.

Schematisch: links für eine Applikation, rechts für ein Applet



Die Gesamtarchitektur für das JDBC und ODBC sieht wie folgt aus:



1.2.2.2. JDBC URLs

Der Pfad auf die Daten wird in JDBC mit Hilfe eines URLs spezifiziert. Der Aufbau der JDBC URLs sieht wie folgt aus:

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

jdbc : die Spezifikation des Protokolles
<subprotocol> : Drivername oder Zugriffsmechanismus, typischerweise odbc falls ein ODBC Treiber eingesetzt werden soll
<subname> : die eigentliche Datenquelle. Falls ODBC eingesetzt wird, dann muss der Subname identisch sein mit der Angabe im ODBC Manager

Falls die Datenbank sich auf einem entfernten Host befindet, dann muss zusätzlich der Hostname als Teil des Subprotokolls angegeben werden:

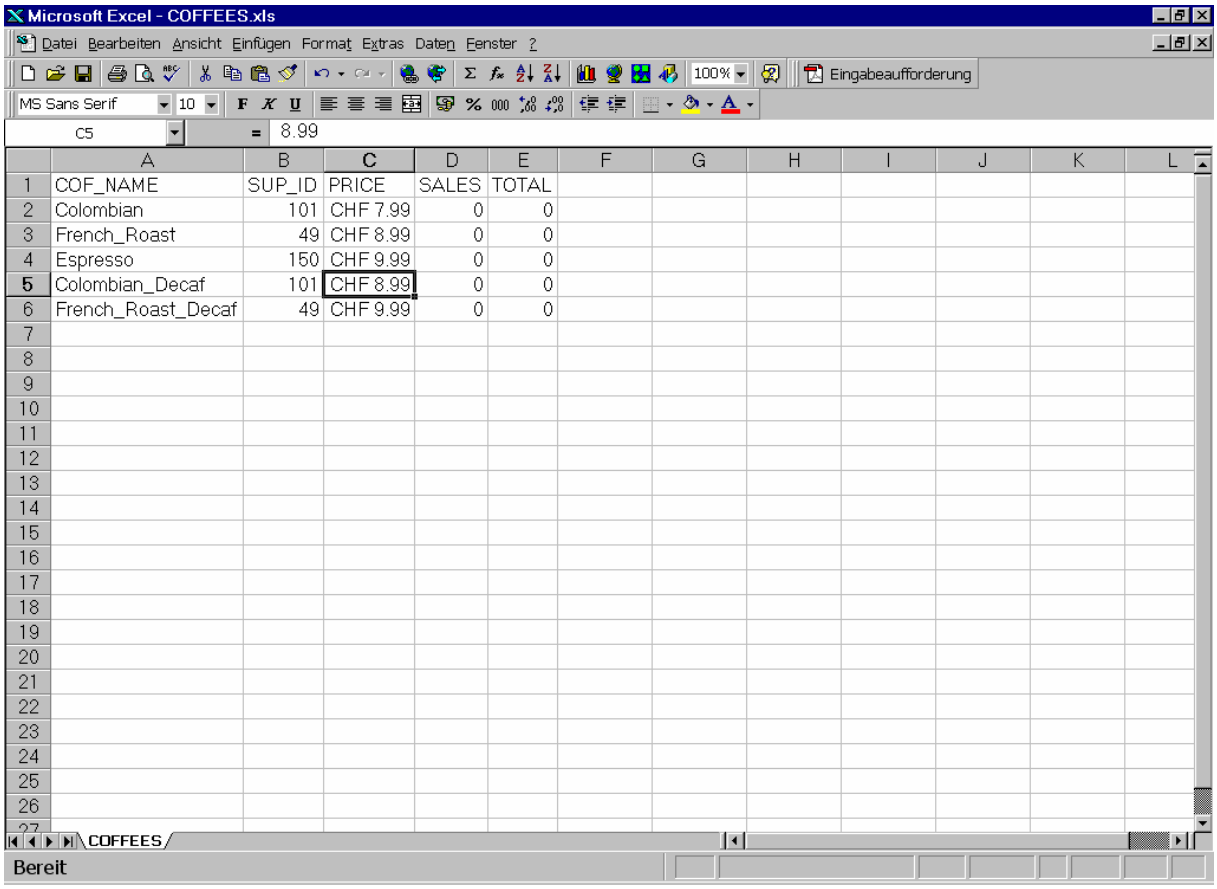
//hostname:port/subsubname

Beispiel : jdbc:dbnet://switch:356/henry

wobei dbnet ein Kommunikationsprotokoll zum remote Host ist (fiktiv!).
Oracle ist ein gutes Beispiel für eine Datenbank, auf die man remote, über das Internet zugreifen kann.

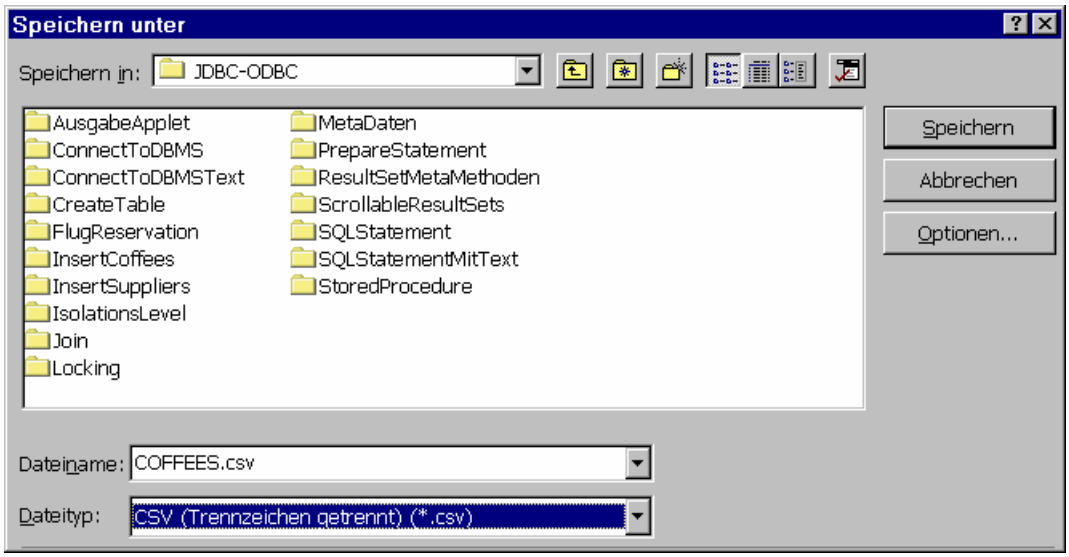
1.2.2.3. Einfache Tests

Als erstes verwenden wir eine einfache Textdatei als Datenbank. Diese Datei definieren wir beispielsweise indem wir in Excel eine Tabelle anlegen:



	A	B	C	D	E
1	COF_NAME	SUP_ID	PRICE	SALES	TOTAL
2	Colombian	101	CHF 7.99	0	0
3	French_Roast	49	CHF 8.99	0	0
4	Espresso	150	CHF 9.99	0	0
5	Colombian_Decaf	101	CHF 8.99	0	0
6	French_Roast_Decaf	49	CHF 9.99	0	0

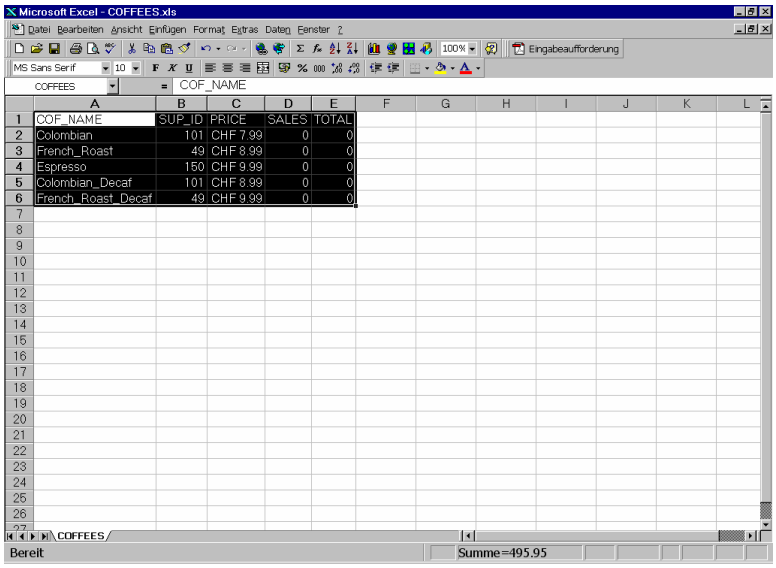
Die Tabelle finden Sie auf dem Server / der CD.
Speichern Sie diese auch noch als Text ab, so dass wir dieselben Daten in ODBC als Textdatei und als Excel Arbeitsblatt definieren können:



1.2.2.4. Definition eines Datenbereiches in Excel als SQL Tabelle

Damit Sie auf einen Bereich in einem Arbeitsblatt in Excel über ODBC zugreifen können, müssen Sie diesen Bereich zuerst benennen:

1) markieren Sie die Daten im Arbeitsblatt von Excel:

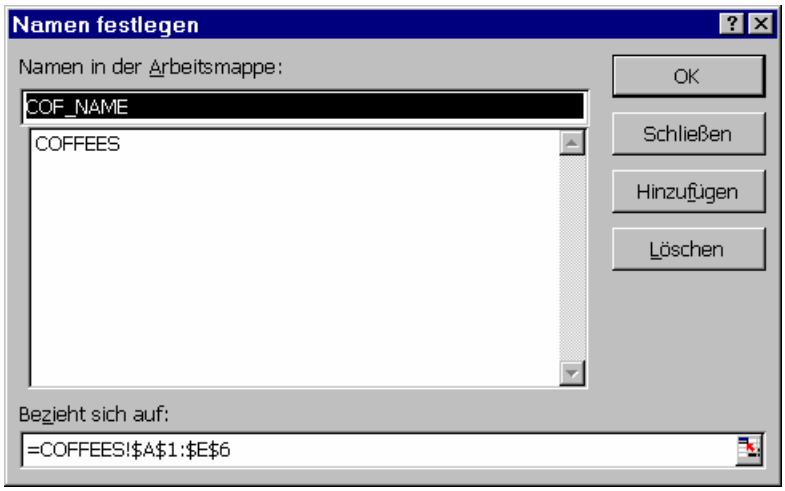


2) benennen Sie diesen Bereich:
   - ganz oben in der Menüliste von Excel finden Sie die Rubriken
     "Datei Bearbeiten Ansicht Einfügen Format Extras Daten Fenster"

   - wählen Sie "Einfügen"
     Sie sehen nun die Optionen
     "Zellen Zeilen ... Funktion Namen Kommentar ..."

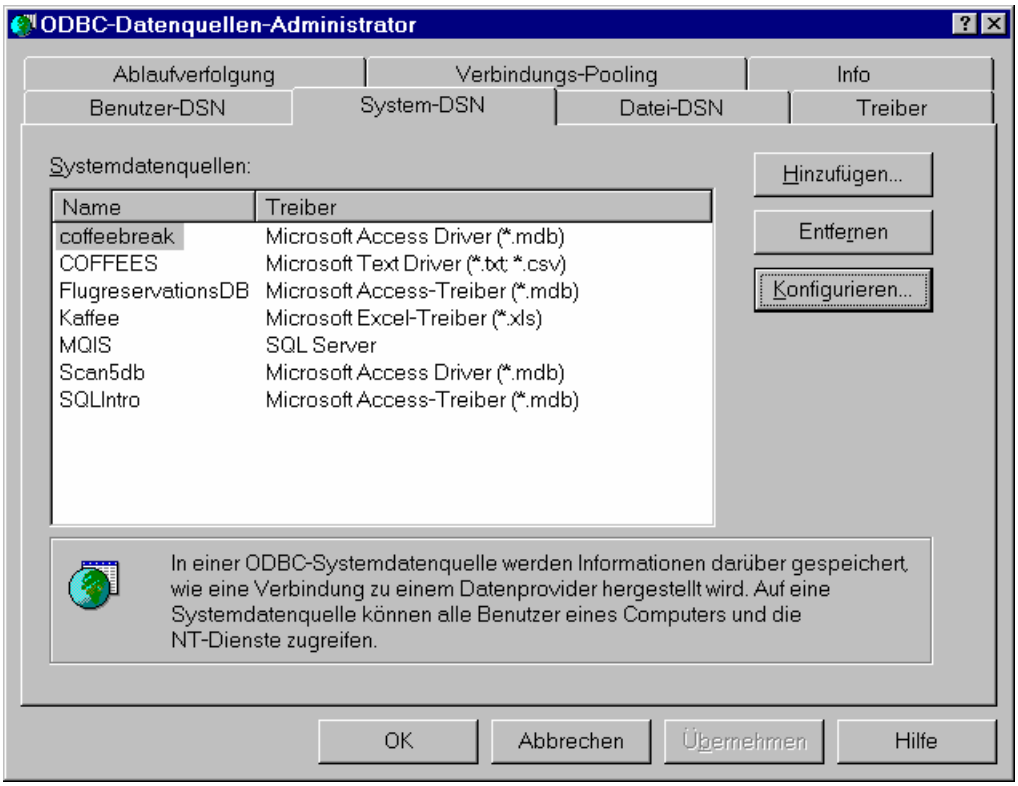
   - wählen Sie "Namen" aus
     Sie sehen nun die Optionen
     "Festlegen Einfügen Erstellen ..."

   - wählen Sie "Festlegen":



Damit haben sie Excel seitig alles definiert, was Sie definieren können und zudem eine Textdatei generiert, welche die selben Daten enthält und auch als Tabelle in ODBC verwendet werden kann.

Binden Sie nun diese Dateien in ODBC ein. Sie gehen dabei genau so vor wie oben beschrieben! In der Systemsteuerung, speziell der ODBC Beschreibung, sollten Sie nun etwa folgende Liste sehen (etwas weniger oder mehr, je nachdem ob Sie selber auch schon andere ODBC Datenquellen definiert haben).



1.2.2.5. Datei, Datenbank undTabellen

Eine Datenbank besteht in der Regel aus einer riesigen Datei in der sich alle Tabellen befinden.

Falls wir eine Textdatei als Datenbanktabelle in ODBC definieren, dann entspricht der Tabelle die Textdatei und gleichzeitig der Datenbank.

Falls Sie ein Excel Arbeitsblatt als ODBC Datenquelle definieren, entspricht der benannte Bereich im Arbeitsblatt der Tabelle.

1.2.3. Einführendes JDBC Beispiel - Verbindungsaufbau

Da Sie nun ODBC und JDBC korrekt aufgesetzt haben, können Sie daran gehen, zu versuchen, auf diese ODBC Datenquellen zuzugreifen.

Beachten Sie:
Sie greifen auf eine ODBC Datenquelle zu! Der Name der Datenquelle in ODBC ist das einzige was Sie wissen müssen, also nicht der Dateiname oder der Tabellenname!
Sie können also die Datenquellen physisch austauschen, ohne an Ihren Programm irgend etwas ändern zu müssen. Ihr Programm sieht nur die ODBC Bezeichnung und hat keine Ahnung was sich dahinter versteckt. Sie könnten also zuerst auf die Textdatei, dann auf Excel, dann auf Access... dann auf Oracle zugreifen, immer mit demselben ODBC Namen! Ihr Programm würde keinen Unterschied sehen. ODBC sorgt dafür, dass die physischen Zugriffe korrekt geleitet werden.

Um sicher zu sein, dass die Installation korrekt ist, greifen wir lediglich auf die Datenquellen zu, lesen aber keine Daten.

```
package connecttodbmstext;

/**
 * Title:        Test
 * @version 1.0
 */

import java.sql.*;

public class ConnectToDBMSText {
      public static void main(String args[]) {
          String url = "jdbc:odbc:COFFEES";
          Connection con;
          System.out.println("Try : Class.forName - Laden des Drivers");
          try {
            Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");
          } catch(java.lang.ClassNotFoundException e) {
            System.err.print("ClassNotFoundException: ");
            System.err.println(e.getMessage());
          }
          System.out.println("Try : DriverManager.getConnection -
Verbindung herstellen");
          try {
            con = DriverManager.getConnection(url,"", "");// User, Passwd
            System.out.println("con.close() - Verbindung schliessen");
            con.close();
          } catch(SQLException ex) {
            System.err.println("SQLException: Verbindungsaufbau " +
ex.getMessage());
          }
      }
}
```

Ausgabe:

```
Try : Class.forName - Laden des Drivers
Try : DriverManager.getConnection - Verbindung herstellen
con.close() - Verbindung schliessen
```

Jetzt schauen wir uns gleich noch den selben Verbindungsaufbau für Excel an:

das Programm ist fast identisch. Es unterscheidet sich geringfügig, weil die alte ODBC Datenquelle nicht überschreiben werden sollte.

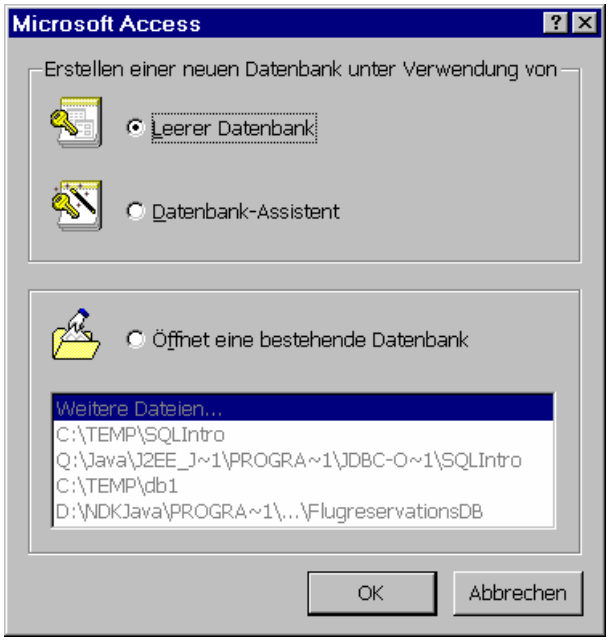
```
public static void main(String args[]) {
    // um die alten ODBC Datenquellen nich zu verlieren
    // habe ich einen anderen ODBC Namen verwendet
    String url = "jdbc:odbc:Kaffees";
    Connection con;
```

Der Rest inklusive der Ausgabe sind völlig identisch.

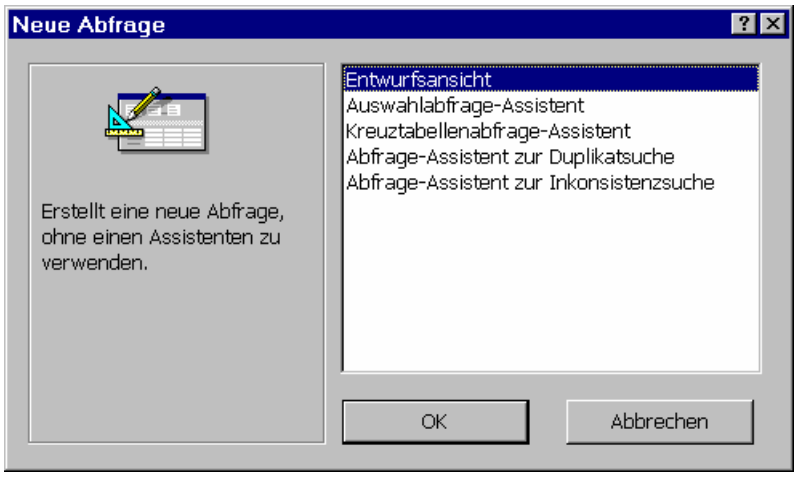
Nun wollen wir einen Schritt weitergehen und mit Access als Datenbank arbeiten. Sie können direkt in Access auch reines SQL verwenden, also ohne die GUI Führung in Access.

1.2.3.1. Access SQL Engine

Öffnen Sie Access und kreieren Sie ohne Wizzard eine Datenbank, beispielsweise IntroSQL.

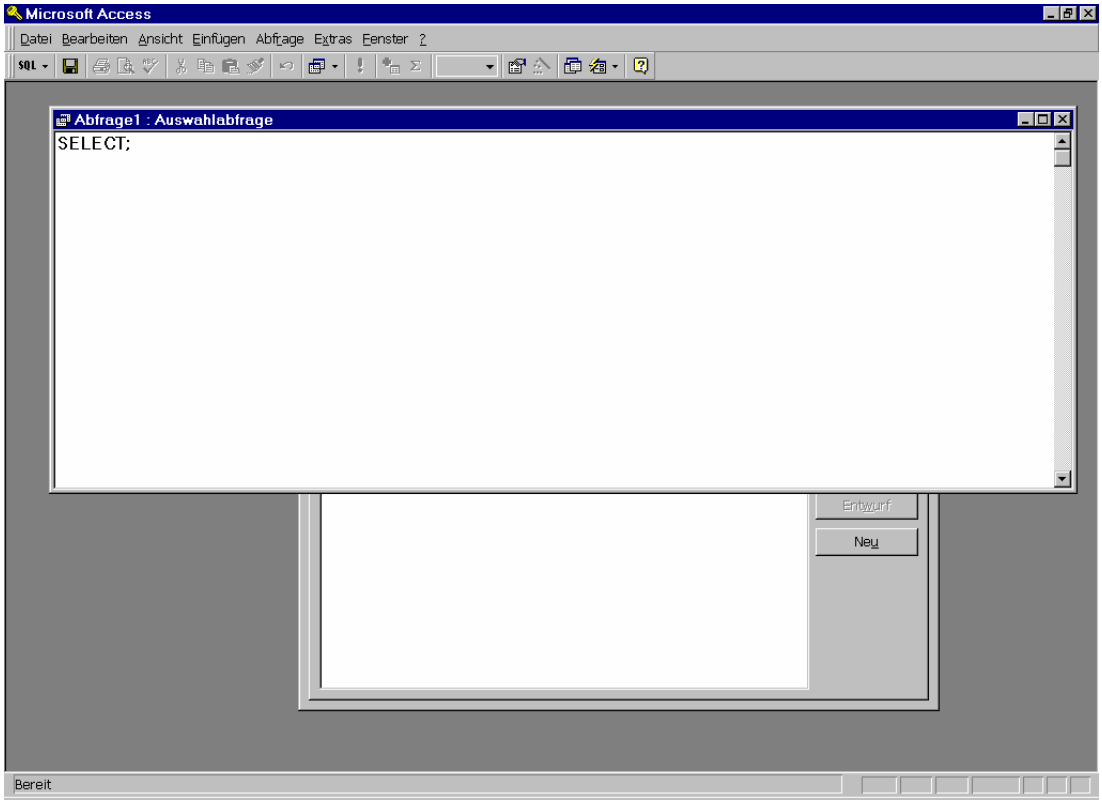


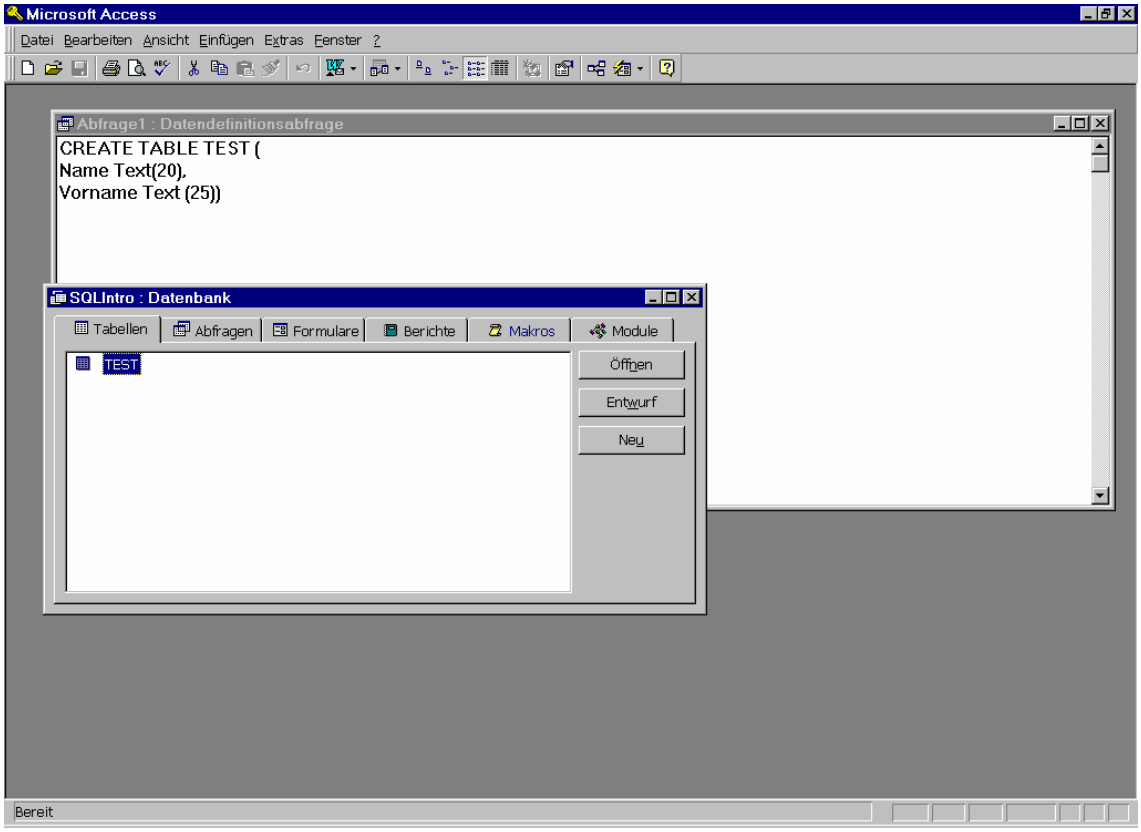
Nun können Sie einfach scheinbar eine Auswertung kreieren (klicken Sie auf neue Abfrage und wählen Sie einfach keinen Wizzard aus).



Wählen Sie nun in der Menüliste die Option "Ansicht" und daraus "SQL":

In dieses Fenster können Sie gültige SQL Befehle eintippen, beispielsweise "CREATE TABLE...." oder "SELECT ..." (siehe weiter unten für eine Beschreibung der SQL Befehle).





Im Folgenden werden wir nun sehen, wie man all dies auch direkt aus Java machen kann.

1.3. Aufsetzen einer Datenbank

Wir werden voraussetzen, dass die Datenbank COFFEEBREAK bereits existiert. Das Kreieren einer Datenbank haben wir eigentlich oben schon gesehen:

- falls Sie eine Textdatei zur Datenbank machen, kreiert der SQL CREATE TABLE Befehl einfach eine neue Datei
- falls Sie ein Excel Arbeitsblatt zur Datenbank gemacht haben, wird mit CREATE TABLE einfach ein neues Arbeitsblatt kreiert

Wir gehen aber davon aus, dass Sie eine vollständigere DBMS verwenden, beispielsweise Access. Sie finden auf dem Server / der CD ein Beispiel.

Eines sollten Sie beachten: unsere Tabellen enthalten sehr wenig Daten. In der Realität wird die Datenmenge sehr gross sein, sonst würde man nicht Datenbanken einsetzen. Dann wird auch die Performance entsprechend viel schlechter!

1.3.1. Szenario

Sie sollen für einen kleinen Kaffeeshop eine Mini-Applikation kreieren. Der Shop verkauft Kaffee verpackt (pfundweise) und offen (tassenweise).

Der Einfachheit halber nehmen wir an, dass der Besitzer des Shops lediglich zwei Tabellen benötigt:

- eine mit den Kaffeesorten
- eine mit den Kaffeelieferanten

Als erstes werden wir auch hier eine Vernindung zur Datenbank herstellen und dann zeigen, wie SQL Befehle an die DBMS gesendet werden können, über JDBC (und ODBC).

Die Ergebnisse werden wir in Java aufbereiten und eventuell formattiert ausgeben.

Sie können die Programme mit jeder gängigen Datenbank, auch einer Textdatei und Excel testen. Sie müssen dazu lediglich eine ODBC Datenquelle definieren können.

1.4. Verbindungsaufbau

Als erstes müssen wir auch in diesem Fall eine Verbindung zur Datenbank aufbauen. Dies besteht aus zwei Schritten:

1) laden des Treibers
2) Verbindungsaufbau.

1.4.1. Laden des Treibers

Das Laden des Treibers oder der Treiber ist recht einfach und wird mit einer einzigen Programmzeile beschrieben. Falls Sie beispielsweise die JDBC-ODBC Bridge einsetzen, sieht diese folgendermassen aus:

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

Falls Sie einen anderen Treiber laden wollen, müssen Sie entsprechende Treiberanweisungen eingeben. Sie finden diese in der Regel in der Dokumentation zur DBMS, speziell zum Thema JDBC Treiber.

Falls der Treiber `jdbc.driverXYZ` heisst, dann müssten Sie folgende Zeile einfügen:

```
Class.forName("jdbc.driverXYZ");
```

Sie kreieren *keine* Instanz des Treibers mit `new`... (obschon dies auch möglich wäre) und registrieren ihn dann beim DriverManager: `Class.forName()` erledigt dies für Sie!

Falls Sie eine eigene Instanz kreieren, wird der Treiber doppelt erfasst, was aber auch nichts macht.

Falls Sie nun den Treiber geladen haben, können wir die Verbindung aufbauen.

1.4.2. Verbindungsaufbau

Der zweite Schritt beim Verbindungsaufbau besteht in der Verbindung zur DBMS durch den Treiber. Dies geschieht in folgender Programmzeile:

```
Connection con = DriverManager.getConnection(url,
                     "meinBenutzername", "meinPasswort");
```

Das Schwierigste am Aufbau ist also die URL festzulegen. Falls Sie die JDBC-ODBC Bridge einsetzen, startet die URL mit `jdbc:odbc`. Der Rest der URL beschreibt ganz allgemein Ihre Datenquelle oder das DBMS. Falls Ihre ODBC Datenquelle "Rigi" heisst, würde Ihre URL vermutlich `jdbc:odbc:Rigi` heissen. Falls Ihr Benutzername 'Weggis' und Ihr Passwort 'Brunnen' heisst, würden Sie also mit

```
String url = "jdbc:odbc:Rigi";
Connection con = DriverManager.getConnection(url, "Weggis",
"Brunnen");
```

eine Verbindung zu Ihrer Datenquelle aufbauen können.

Falls Sie einen Treiber einer 3rd Party einsetzen, werden Sie in deren Dokumentation die genaue Beschreibung der URL und weitere Angaben finden.

Beispielsweise könnte es sein, dass die DBMS oder der Treiber ein eigenes Subprotokoll "pizza" verwendet. In diesem Fall würde die URL entsprechend angepasst werden müssen:

```
jdbc:pizza:...
```

Falls einer der Treiber, die Sie geladen haben (mit `Class.forName()`) eine der URLs in `DriverManager.getConnection(url...)` versteht, wird der Driver Manager eine Verbindung zur DBMS aufbauen, welche in der URL spezifiziert wurde. Der Driver-Manager kümmert sich um alle Details und baut ohne Ihr weiteres Hinzutun die Verbindung zur DBMS auf.

Falls Sie selber keinen eigenen Treiber schreiben wollen, dann brauchen Sie kaum weitere Methoden aus dem Interface `Driver`. Die einzige Methode, die Sie kennen müssen ist für diese Klasse also `getConnection(...)`.

Die Methode liefert Ihnen eine offene Verbindung zur DBMS und gestattet Ihnen SQL Anweisungen an die DBMS zu senden.

1.5. Aufsetzen der Tabellen

1.5.1. Kreieren einer Tabelle

Als erstes kreieren wir die Tabellen für unser Beispiel. In der Tabelle COFFEES steht die Information über verkauften Kaffee im Shop. Diese Tabelle enthält den Kaffeenamen, den Preis des Kaffees pro Pfund, die Anzahl Pfundpackungen, die verkauft wurden und das Total (Verkauf +Ausschank):

COF_NAME	*SUP_ID*	*PRICE*	*SALES*	*TOTAL*
Colombian	101	7.99	0	0
French_Roast	49	8.99	0	0
Espresso	150	9.99	0	0
Colombian_Decaf	101	8.99	0	0
French_Roast_Decaf	49	9.99	0	0

Wir werden die Tabelle später in SQL beschreiben. Für den Moment geht es nur darum eine Idee über deren Aufbau und Inhalt zu erhalten. Die zwei letzten Felder können wir später festlegen.

Die erste Spalte beschreibt die Kaffeesorte (COF_NAME). Diese Spalte muss Daten vom SQL Datentyp VARCHAR der maximalen Länge 32 Zeichen aufnehmen können. Die Kaffeesorten sind eindeutig und unterschiedlich. Wir werden also diese Spalte als Schlüssel verwenden.

Die zweite Spalte (SUP_NAME) enthält einen Code, eine Ziffer, mit der der Kaffeelieferant bezeichnet wird. Diese Spalte enthält Daten vom SQL Datentyp INTEGER.

Die dritte Spalte (PRICE) speichert den Preis des Kaffees. Diese Daten sind vom SQL Datentyp FLOAT, weil sie einen Dezimalpunkt enthalten. In produktiven Beispielen würden Sie wahrscheinlich den SQL Datentyp DECIMAL oder NUMERIC verwenden.

Die Spalte SALES speicher die verkaufszahlen, also SQL Datentypen INTEGER.

Die letzte Spalte SALES ist ebenfalls vom Typ INTEGER, weil es die Gesamtzahl Kaffee enthält, der am Tage verkauft wurde (Pfundpackungen plus offener Ausschank).

Die SUPPLIERS , die zweite Tabelle in unserer Datenbank, enthält Informationen über jeden der Lieferanten:

SUP_ID	SUP_NAME	STREET	CITY	STATE	ZIP
101	Hag	Marktgasse 12	Bern	BE	3000
49	Tschibo	Limmatquai 78	Zuerich	ZH	8000
150	Jacobs	Zwinglistrasse 12	Luzern	LU	6000

Die Tabellen COFFEES und SUPPLIERS enthalten beide eine Spalte SUP_ID. Damit kann man beide Tabellen in einer SELECT Anweisung kombinieren. In der Tabelle SUPPLIERS ist diese Spalte der Primärschlüssel, also eindeutig definiert. In der Tabelle COFFEES ist SUP_ID einfach eine Spalte, ein Fremdschlüssel um genau zu sein. Ein Fremdschlüssel ist ein Primärschlüssel, welcher aus einer anderen Tabelle importiert wird, also ein Feld, das in einer anderen Tabelle Primärschlüssel ist.

In der Tabelle SUPPLIERS erscheint jeder Wert von SUP_ID lediglich einmal, da es sich um einen eindeutigen Primärschlüssel handelt.

In der Tabelle CAFFEES dagegen können die Werte des Fremdschlüssels mehrfach auftreten. Das entspricht auch der Realität: ein Kaffeelieferant kann an mehrere Kunden liefern. Er kann aber auch mehrere Kaffeesorten liefern.

Der Einfachheit halber nehmen wir an, dass die Kaffeesorten jeweils nur von einem Supplier stammen können.

In der folgenden SQL Anweisung kreieren wir die Tabelle CAFFEES. Die Einträge innerhalb der äusseren Klammer bestehen aus einem Spaltennamen gefolgt von einem Leerzeichen und einem SQL Datentyp, mit dem beschrieben wird, was (typenmässig) in der Spalte abgespeichert werden soll.
Die einzelnen Spalten werden durch ein Komma getrennt.
Ein Typ VARCHAR entspricht variablem Text. In der Klammer dahinter steht die Länge des Feldes.

In der SQL Anweisung wird angegeben, dass der Name der Spalte COF_NAME sein soll und diese maximal 32 Zeichen breit sein darf. Falls der Text länger ist, wird eine Exception geworfen.

```
CREATE TABLE COFFEES (
        COF_NAME VARCHAR(32),
        SUP_ID INTEGER,
        PRICE FLOAT, SALES INTEGER,
        TOTAL INTEGER
)
```

Beachten Sie, dass nach der SQL Anweisung kein Abschlusszeichen (beispielsweise ein Semikolon ;) steht. Die Abschlusszeichen unterscheiden sich je nach DBMS und JDBC bzw. ODBC vereinheitlichten diesen Abschluss einer Anweisung, um die Programme möglichst universell gestalten zu können. Oracle verwendet ';', Sybase "go", ...

Eine weitere Bemerkung:
die SQL Schlüsselwörter sind gross geschrieben, wegen der Lesbarkeit, nicht aus syntaktischen Gründen! Gemäss SQL Standard sind die Schlüsselwörter nicht casesensitiv. Die folgende Anweisung ist also eine legale SQL Anweisung:

```
SELECT First_Name, Last_Name
FROM Employees
WHERE Last_Name LIKE "Washington"
```

genauso wie diese:

```
select First_Name, Last_Name from Employees where
Last_Name like "Washington"
```

Anders sieht es mit den Werten aus: gemäss obiger Abfrage wollen Sie 'Washington', nicht 'washington' als Namen finden. In diesen Fällen wird die Gross- und Kleinschreibung also strikt beachtet.

Je nach DBMS kann man auch die Spaltennamen schreiben. Einige DBMS verlangen, dass der Spaltenname genau so aussieht, wie im CREATE Statement. bei anderen wiederum spielt dies überhaupt keine Rolle.

Aus Sicherheitsgründen werden wir daher einfach Grossbuchstaben einsetzen, überall.

Um unsere SQL Anweisung im Java Programm verwenden zu können, definieren wir die Anweisung als String und übergeben Sie im Java Programm an eine Abfragemethode.

```
String createTableCoffees = "CREATE TABLE COFFEES " +
"(COF_NAME VARCHAR(32), SUP_ID INTEGER, PRICE FLOAT, " +
"SALES INTEGER, TOTAL INTEGER)";
```

Die Datentypen, die wir in der CREATE TABLE Anweisung benutzten, sind generische SQL Datentypen. Diese sind in der Klasse `java.sql.Types` definiert.

Welche Datentypen in dieser Definition enthalten sind, sehen Sie auf der nächsten Seite, einem Auszug aus der API Dokumentation von JDBC 2.0:

http://java.sun.com/j2se/1.3/docs/api/java/sql/Types.html

Field Summary static int **ARRAY**
JDBC 2.0 A type representing an SQL ARRAY. static int **BIGINT**
static int **BINARY**
static int **BIT**
static int **BLOB**
JDBC 2.0 A type representing an SQL Binary Large Object.
static int **CHAR**
static int **CLOB**
JDBC 2.0 A type representing an SQL Character Large Object.
static int **DATE**
static int **DECIMAL**
static int **DISTINCT**
JDBC 2.0 A type based on a built-in type.
static int **DOUBLE**
static int **FLOAT**
static int **INTEGER**
static int **JAVA_OBJECT**
JDBC 2.0 A type representing a Java Object.
static int **LONGVARBINARY**
static int **LONGVARCHAR**
static int **NULL**
static int **NUMERIC**
static int **OTHER**
OTHER indicates that the SQL type is database-specific and gets mapped to a Java object that can be accessed via the methods getObject and setObject.
static int **REAL**
static int **REF**
JDBC 2.0 A type representing an SQL REF.
static int **SMALLINT**
static int **STRUCT**
JDBC 2.0 A type consisting of attributes that may be any type.
static int **TIME**
static int **TIMESTAMP**
static int **TINYINT**
static int **VARBINARY**
static int **VARCHAR**

Bevor wir die SQL Anweisung ausführen, gehen wir noch genauer auf den Ablauf einer JDBC Verbindung ein.

1.5.2. Kreieren von JDBC Anweisungen

Ein `Statement` Objekt sendet Ihre SQL Anweisung an die DBMS. Sie kreieren einfach ein `Statement` Objekt und führen es aus, unter Ausnutzen der passenden Methode.

SELECT Anweisungen können Sie mit `execute`**`Query`**`()` ausführen.
CREATE, INSERT, DROP und allfällig weitere Anweisungen, mit denen Tabellen modifiziert oder kreiert werden, ruft man mit der `execute`**`Update`**`()` Methode auf.

Ein `Statement` Objekt können Sie mittels einer aktiven `Connection` kreieren:

1) Kreieren der Anweisung:
```
Statement stmt = con.createStatement();
```
2) Übergeben der SQL Anweisung und Ausführen der Anweisung:
```
stmt.executeUpdate("CREATE TABLE COFFEES " +
 "(COF_NAME VARCHAR(32), SUP_ID INTEGER, PRICE FLOAT, " +
"SALES INTEGER, TOTAL INTEGER)");
```
3) alternativ: `stmt.executeUpdate(createTableCoffees);`

1.5.3. Ausführende Anweisungen

Da wir in unserer Anweisung eine Tabelle kreieren wollen, müssen wir die `executeUpdate()` Methode einsetzen. Die CREATE TABLE Anweisung ist eine DDL (data definition language) Anweisung. Anweisungen, welche Tabellen kreieren, verändern oder löschen sind Beispiele für DDL Anweisungen, die mit der `executeUpdate()` Methode ausgeführt werden.

Zudem wird diese Methode wie der Name sagt auch für Datenmutationen in einer Tabelle eingesetzt.

1.5.4. Daten in die Tabelle einfügen

Nun wissen wir, wie die Tabelle kreiert werden kann: durch Angabe der Spaltennamen und der Datentypen. Aber dies ist nur einer der Schritte! Die Tabelle enthält noch keine Daten. Übrigens: die Spaltenreihenfolge bleibt die selbe wie beim Kreieren der Tabelle.

Im folgenden SQL Programmcode fügen wir die Daten in die Tabelle ein.

Beispiel:
COF_NAME: Colombian
SUP_ID: 101
PRICE: 7.99
SALES: 0
TOTAL 0

Da das Programm noch nicht aktiv ist, sind die Verkaufszahlen noch null.
Diese Daten lassen sich mit einer INSERT SQL Anweisung in die Tabelle der Datenbank einfügen. In Java sieht dies folgendermassen aus:
.

```
Statement stmt = con.createStatement();
stmt.executeUpdate(
    "INSERT INTO COFFEES " +
    "VALUES ('Colombian', 101, 7.99, 0, 0)");
```

Beachten Sie, dass die Zeichenkette nicht auf einer Zeile Platz hat und daher mit "..." + "..." zwei Teilzeichenketten verknüpft werden.

Beachten Sie zudem, dass der Text in Spalte COF_NAME in einfachen Anführungszeichen eingegeben wird.

Die Eingabe der weiteren Daten ist nun eigentlich keine grosse Sache mehr: wir können einfach diese Zeile kopieren und die Daten modifizieren.

```
stmt.executeUpdate("INSERT INTO COFFEES " +
    "VALUES ('French_Roast', 49, 8.99, 0, 0)");
```

Und hier ist der Rest:

```
stmt.executeUpdate("INSERT INTO COFFEES " +
    "VALUES ('Espresso', 150, 9.99, 0, 0)");
stmt.executeUpdate("INSERT INTO COFFEES " +
    "VALUES ('Colombian_Decaf', 101, 8.99, 0, 0)");
stmt.executeUpdate("INSERT INTO COFFEES " +
    "VALUES ('French_Roast_Decaf', 49, 9.99, 0, 0)");
```

Das gesamte Programm sieht damit folgendermassen aus:

```
package createtable;

import java.sql.*;

public class CreateCoffees {

      public static void main(String args[]) {

            String url = "jdbc:odbc:coffeebreak";
            Connection con;
            String createString;
            createString = "create table COFFEES " +
                                          "(COF_NAME varchar(32), " +
                                          "SUP_ID INTEGER, " +
                                          "PRICE FLOAT, " +
                                          "SALES INTEGER, " +
                                          "TOTAL INTEGER)";
            Statement stmt;

            try {
                 System.out.println("Class.forName - Driver laden");
                  Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

            } catch(java.lang.ClassNotFoundException e) {
                  System.err.print("ClassNotFoundException: ");
                  System.err.println(e.getMessage());
            }

            try {
                  System.out.println("DriverManager.getConnection -
                                                Verbindung aufbauen");
                  con = DriverManager.getConnection(url,
                                                        "", "");
                  System.out.println("createStatement -
                                    SQL Cursor zur Datenquelle kreieren");
                  stmt = con.createStatement();
            try {
                  System.out.println("executeUpdate - SQL Statement
                                    uebergeben");
                  stmt.executeUpdate(createString); }
            catch(SQLException eSQL) {
                  System.out.println("executeUpdate - Fehler beim Kreieren
                        der Tabelle "+eSQL.getMessage() );
            }

            System.out.println("close() - Verbindung abbauen");
            stmt.close();
            con.close();

            } catch(SQLException ex) {
                  System.err.println("SQLException: " + ex.getMessage());
            }
      }
}
```

1.5.5. Lesen von Daten aus der Tabelle

Nachdem wir nun Daten in die Tabelle eingefügt haben, können wir mit einer SELECT Anweisung diese Werte wieder aus der Tabelle herauslesen.

Da wir einfach alle Spalten (also Felder in der Tabelle) lesen wollen, können wir einfach einen "*" für "alle Spalten" verwenden. Wir schränken die Daten auch innerhalb der Spalten nicht ein: die WHERE Anweisung wird ebenfalls weggelassen.

```
SELECT * FROM COFFEES
```

Das Ergebnis ist eine Auflistung der gesamten Tabelle:

```
COF_NAME          SUP_ID  PRICE   SALES   TOTAL
---------------   ------  -----   -----   -----
Colombian         101     7.99    0       0
French_Roast       49     8.99    0       0
Espresso          150     9.99    0       0
Colombian_Decaf   101     8.99    0       0
French_Roast_Decaf 49     9.99    0       0
```

Sie können diese Abfrage beispielsweise direkt in einem SQL Fenster in Access ausführen oder eine schicke Auswertung definieren und starten. Unten werden wir sehen, wie die Daten in Java gelesen und ins Programm übernommen werden können.

Falls wir nicht alle Spalten abfragen wollen, können wir einfach die gewünschten Spalten angeben:

```
SELECT COF_NAME, PRICE FROM COFFEES
```

Als Ergebnis erhalten wir:

```
COF_NAME          PRICE
--------          ----------
Colombian         7.99
French_Roast      8.99
Espresso          9.99
Colombian_Decaf   8.99
French_Roast_Decaf 9.99
```

Diese SELECT Anweisung generiert die Namen und Preise aller Kaffeesorten in der Tabelle. Nun eine Anweisung, bei der zusätzlich nur die billigen Kaffees aufgelistet werden sollen:

```
SELECT COF_NAME, PRICE
FROM COFFEES
WHERE PRICE < 9.00
```

Das Ergebnis sieht nun folgendermassen aus:

```
COF_NAME          PRICE
--------          -------
Colombian         7.99
French_Roast      8.99
Colombian Decaf   8.99
```

1.5.5.1. Lesen von Daten aus dem ResultSet

Bisher haben wir lediglich die SQL Anweisungen kennen gelernt, wie man die Daten manipulieren kann, aber nicht wie man die Daten auch ins Java Programm übernimmt.

In JDBC werden die Daten, die aus einer Abfrage resultieren, in einem `ResultSet` Objekt an das aufrufende Programm geliefert. Daher müssen wir ein solches Objekt in unserem Programm deklarieren. Ein Beispielprogramm könnte folgendermassen aussehen:

```
ResultSet rs = stmt.executeQuery(
                  "SELECT COF_NAME, PRICE FROM COFFEES");
```

1.5.5.2. Einsatz der Methode `next()`

Die Variable rs, die Instanz der ResultSet Klasse, enthält die Zeilen, welche aus der Datenbank gelesen wurden. Damit wir auf diese Daten zugreifen können, müssen wir Zeile für Zeile durch das ResultSet gehen und die Daten gemäss ihrem Datentyp herauslesen.

Die Methode `next()` liefert jeweils den nächsten Datensatz aus dem `ResultSet`. Dabei wird ein sogenannter Cursor eingesetzt: dieser zeigt vor dem Lesen des ersten Datensatzes auf die Position *vor* dem ersten Datensatz. Somit kann man direkt mit einer Abfrageschleife starten und alle Daten herausholen.

```
String query = "SELECT COF_NAME, PRICE FROM COFFEES";
ResultSet rs = stmt.executeQuery(query);
while (rs.next()) {
    String s = rs.getString("COF_NAME");
    float n = rs.getFloat("PRICE");
    System.out.println(s + "   " + n);
}
```

Ab JDBC 2.0 kann man den Cursor vorwärts und rückwärts bewegen, inJDBC 1.0 lediglich vorwärts (`next()`).

1.5.5.3. Einsatz den `getXXX()` Methoden

Mit der `getXXX()` Methode zum passenden Datentyp kann man die Daten zu den jeweiligen Salten aus dem aktuellen Datensatz im Resultset bestimmen.

In unserem Fall besteht die Spalte COF_NAME aus Text vom SQL Datentyp VARCHAR. In Java entspricht dies einfach einer String Variable. Die entsprechende Methode ist getString().

In der zweiten Spalte stehen Daten vom SQL Datentyp FLOAT. In Java entspricht dies dem gleichen Datentyp. Die entprechende Methode ist getFloat().

Und hier ein Beispielfragment für die Abfrage eines Resultsets:

```
String query = "SELECT COF_NAME, PRICE FROM COFFEES";
ResultSet rs = stmt.executeQuery(query);
while (rs.next()) {
    String s = rs.getString("COF_NAME");
    float n = rs.getFloat("PRICE");
    System.out.println(s + "   " + n);
}
```

Als Ausgabe erhalten wir:

```
Colombian               7.99
French_Roast            8.99
Espresso                9.99
Colombian_Decaf         8.99
French_Roast_Decaf      9.99
```

Schauen wir uns nun diese `getXXX()` Methoden noch einmal etwas genauer an:

```
String s = rs.getString("COF_NAME");
```

Die Methode `getString()` wird auf das Objekt `rs` der `ResultSet` Klasse angewandt,. `getString()` liest (`get`) jenen Wert, der in der Spalte COF_NAME in der aktuellen Zeile des Resultsets vorhanden ist.

`getString()` liest einen SQL VARCHAR Wert und konvertiert diesen in eine String Variable in Java. Analog verhält es sich mit der `getFloat()` Methode und der Spalte PRICE vom SQL FLOAT. JDBC offeriert zwei Wege, um die Spalten zu bestimmen, welche in einer Tabelle oderin einem Resultset vorhanden sind. Zum einen können Sie einfach den Spaltennamen als Parameter angeben und die Daten bestimmen. Zum anderen können Sie mit einem Spaltenindex arbeiten: die Spalten werden von 1 bis ... durchnummeriert. Mit der zweiten Technik würde unsere Abfrage folgendermassen aussehen:

```
String s = rs.getString(1);
float n = rs.getFloat(2);
```

Die erste Zeile bestimmt den Wert der ersten Spalte, eine Zeichenkette (COF_NAME) und konvertiert diese in eine String Variable. Die zweite Zeile liest den Wert aus der zweiten Spalte der aktuellen Zeile im ResultSet, konvertiert diesen in ein Java Float und übergibt den Wert an die Java Variable `n`. Die Spaltennummer bezieht sich auf die Position im Resultset, nicht in der Tabelle.Der Einsatz der Spaltennummer ist sicher effizienter, weil Sie einfach Schleifen bauen können, um die Variablenwerte zu bestimmen.

JDBC ist flexibel: die `getXXX()` Methoden erlauben es in der Regel auch Datentypen zu lesen, welche nicht genau dem Datentyp XXX entsprechen. `getInt()` kann beliebige nummerische oder zeichenorientierte Datentypen lesenn: BINARY , VARBINARY , LONGVARBINARY , DATE , TIME oder TIMESTAMP .

Zusammenfassung: `getXXX()` Methoden

SQL3 type	*getXXX method*	*setXXX method*	*updateXXX method*
BLOB	getBlob	setBlob	updateBlob
CLOB	getClob	setClob	updateClob
ARRAY	getArray	setArray	updateArray
Structured type	getObject	setObject	updateObject
REF (structured type)	getRef	setRef	updateRef

1.5.5.4. Einsatz der `getString()` Methode

Obschon die Methode `getString()` eigentlich für die SQL Datentypen CHAR und VARCHAR bestimmt ist, kann man mit dieser Methode alle SQL Basisdatentypen lesen, ausser den neuen in SQL3 definierten (BLOB =binary large objects und ähnliche). Auf die neuen SQL 3 Datentypen kommen wir später noch zurück.

Das Lesen der Daten mit getString() ist sehr praktisch, da man damit universell einsetzbare Programme schreiben kann. Allerdings hat die Meethode auch Nachteile: falls Sie nummerische Daten damit lesen, werden diese in ein String Objekt umgewandelt und Sie müssen dann daraus den nummerischen Wert wieder herauslesen. Falls Sie jedoch den Wert als String einsetzen wollen, besteht kaum ein Nachteil.

1.5.5.5. Einsatz der `ResultSet.getXXX()` Methoden zum Lesen der JDBC Datentypen

	TINYINT	SMALLINT	INTEGER	BIGINT	REAL	FLOAT	DOUBLE	DECIMAL	NUMERIC	BIT	CHAR	VARCHAR	LONGVARCHAR	BINARY	VARBINARY	LONGVARBINARY	DATE	TIME	TIMESTAMP
getByte	Y	x	x	x	x	x	x	x	x	x	x	x	x						
getShort	x	Y	x	x	x	x	x	x	x	x	x	x	x						
getInt	x	x	Y	x	x	x	x	x	x	x	x	x	x						
getLong	x	x	x	Y	x	x	x	x	x	x	x	x	x						
getFloat	x	x	x	x	Y	x	x	x	x	x	x	x	x						
getDouble	x	x	x	x	x	Y	Y	x	x	x	x	x	x						
getBigDecimal	x	x	x	x	x	x	x	Y	Y	x	x	x	x						
getBoolean	x	x	x	x	x	x	x	x	x	Y	x	x	x						
getString	x	x	x	x	x	x	x	x	x	x	Y	Y	x	x	x	x	x	x	x
getBytes														Y	Y	x			
getDate											x	x	x				Y		x
getTime											x	x	x					Y	x
getTimestamp											x	x	x				x	x	Y
getAsciiStream											x	x	Y	x	x	x			
getUnicodeStream											x	x	Y	x	x	x			
getBinaryStream														x	x	Y			
getObject	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

Ein "x" zeigt an, dass die getXXX() Methode eingesetzt werden kann, um diesen JDBC Datentyp zu lesen.

Ein " Y " zeigt an, dass die getXXX() Methode eingesetzt werden kann und die empfohlene Methode ist, um diesen JDBC Datentyp zu lesen.

Da das Lesen der Tabelle etwas schwierig sein kann, sehen Sie hier gleich noch zwei Alternativen:
- die erste Version startet mit den JDBC Datentypen
- die zweite startet mit den Resultset getXXX Methoden.

1.5.5.5.1. Lesen der JDBC Datentypen

Wir starten mit den JDBC Datentypen und geben die dazupassenden Methoden an.

TINYINT: getByte (empfohlene Methode)

dieser Datentyp kann auch mit folgenden Methoden: getShort, getInt, getLong, getFloat, getDouble, getBigDecimal, getBoolean, getString, getObject

SMALLINT: getShort (empfohlene Methode)

dieser Datentyp kann auch mit folgenden Methoden: getByte, getInt, getLong, getFloat, getDouble, getBigDecimal, getBoolean, getString, getObject

INTEGER: getInt (empfohlene Methode)

dieser Datentyp kann auch mit folgenden Methoden: getByte, getShort, getLong, getFloat, getDouble, getBigDecimal, getBoolean, getString, getObject

BIGINT: getLong (empfohlene Methode)

dieser Datentyp kann auch mit folgenden Methoden: getByte, getShort, getInt, getFloat, getDouble, getBigDecimal, getBoolean, getString, getObject

REAL: getFloat (empfohlene Methode)

dieser Datentyp kann auch mit folgenden Methoden: getByte, getShort, getInt, getLong, getDouble, getBigDecimal, getBoolean, getString, getObject

FLOAT: getDouble (empfohlene Methode)

dieser Datentyp kann auch mit folgenden Methoden: getByte, getShort, getInt, getLong, getFloat, getBigDecimal, getBoolean, getString, getObject

DOUBLE: getDouble (empfohlene Methode)

dieser Datentyp kann auch mit folgenden Methoden: getByte, getShort, getInt, getLong, getFloat, getBigDecimal, getBoolean, getString, getObject

DECIMAL: getBigDecimal (empfohlene Methode)

dieser Datentyp kann auch mit folgenden Methoden: getByte, getShort, getInt, getLong, getFloat, getDouble, getBoolean, getString, getObject

NUMERIC: getBigDecimal (empfohlene Methode)

dieser Datentyp kann auch mit folgenden Methoden: getByte, getShort, getInt, getLong, getFloat, getDouble, getBoolean, getString, getObject

BIT: getBoolean (empfohlene Methode)

dieser Datentyp kann auch mit folgenden Methoden: getByte, getShort, getInt, getLong, getFloat, getDouble, getBigDecimal, getString, getObject

CHAR: getString (empfohlene Methode)

dieser Datentyp kann auch mit folgenden Methoden: getByte, getShort, getInt, getLong, getFloat, getDouble, getBigDecimal, getBoolean, getDate, getTime, getTimestamp, getAsciiStream, getUnicodeStream, getObject

VARCHAR: getString (empfohlene Methode)

dieser Datentyp kann auch mit folgenden Methoden: getByte, getShort, getInt, getLong, getFloat, getDouble, getBigDecimal, getBoolean, getDate, getTime, getTimestamp, getAsciiStream, getUnicodeStream, getObject

LONGVARCHAR: getAsciiStream, getUnicodeStream (empfohlene Methoden)

dieser Datentyp kann auch mit folgenden Methoden: getByte, getShort, getInt, getLong, getFloat, getDouble, getBigDecimal, getBoolean, getString, getDate, getTime, getTimestamp, getObject

BINARY: getBytes (empfohlene Methode)

dieser Datentyp kann auch mit folgenden Methoden: getString, getAsciiStream, getUnicodeStream, getBinaryStream, getObject

VARBINARY: getBytes (empfohlene Methode)

dieser Datentyp kann auch mit folgenden Methoden: getString, getAsciiStream, getUnicodeStream, getBinaryStream, getObject

LONGVARBINARY: getBinaryStream (empfohlene Methode)

dieser Datentyp kann auch mit folgenden Methoden: getString, getBytes, getAsciiStream, getUnicodeStream, getObject

DATE: getDate (empfohlene Methode)

dieser Datentyp kann auch mit folgenden Methoden: getString, getTimestamp, getObject

TIME: getTime (empfohlene Methode)

dieser Datentyp kann auch mit folgenden Methoden: getString, getTimestamp, getObject

TIMESTAMP: getTimestamp (empfohlene Methode)

dieser Datentyp kann auch mit folgenden Methoden: getString, getDate, getTime, getObject

1.5.5.5.2. Welche Datentypen kann ResultSet.getXXX lesen

Jetzt starten wir mit den Methoden und geben die JDBC Datentypen an, welche damit gelesen werden können.

getByte: TINYINT (empfohlene Methode)

kann auch folgende JDBC Datentypen lesen: SMALLINT, INTEGER, BIGINT, REAL, FLOAT, DOUBLE, DECIMAL, NUMERIC, BIT, CHAR, VARCHAR, LONGVARCHAR

getShort: SMALLINT (empfohlene Methode)

kann auch folgende JDBC Datentypen lesen: TINYINT, INTEGER, BIGINT, REAL, FLOAT, DOUBLE, DECIMAL, NUMERIC, BIT, CHAR, VARCHAR, LONGVARCHAR

getInt: INTEGER (empfohlene Methode)

kann auch folgende JDBC Datentypen lesen: TINYINT, SMALLINT, BIGINT, REAL, FLOAT, DOUBLE, DECIMAL, NUMERIC, BIT, CHAR, VARCHAR, LONGVARCHAR

getLong: BIGINT (empfohlene Methode)

kann auch folgende JDBC Datentypen lesen: TINYINT, SMALLINT, INTEGER, REAL, FLOAT, DOUBLE, DECIMAL, NUMERIC, BIT, CHAR, VARCHAR, LONGVARCHAR

getFloat: REAL (empfohlene Methode)

kann auch folgende JDBC Datentypen lesen: TINYINT, SMALLINT, INTEGER, BIGINT, FLOAT, DOUBLE, DECIMAL, NUMERIC, BIT, CHAR, VARCHAR, LONGVARCHAR

getDouble: FLOAT, DOUBLE (empfohlene Methoden)

kann auch folgende JDBC Datentypen lesen: TINYINT, SMALLINT, INTEGER, BIGINT, REAL, DECIMAL, NUMERIC, BIT, CHAR, VARCHAR, LONGVARCHAR

getBigDecimal: DECIMAL, NUMERIC (empfohlene Methoden)

kann auch folgende JDBC Datentypen lesen: TINYINT, SMALLINT, INTEGER, BIGINT, REAL, FLOAT, DOUBLE, BIT, CHAR, VARCHAR, LONGVARCHAR

getBoolean: BIT (empfohlene Methode)

kann auch folgende JDBC Datentypen lesen: TINYINT, SMALLINT, INTEGER, BIGINT, REAL, FLOAT, DOUBLE, DECIMAL, NUMERIC, CHAR, VARCHAR, LONGVARCHAR

getString: CHAR, VARCHAR (empfohlene Methoden)

kann auch folgende JDBC Datentypen lesen: TINYINT, SMALLINT, INTEGER, BIGINT, REAL, FLOAT, DOUBLE, DECIMAL, NUMERIC, BIT, LONGVARCHAR, BINARY, VARBINARY, LONGVARBINARY, DATE, TIME, TIMESTAMP

getBytes: BINARY, VARBINARY (empfohlene Methoden)

kann auch folgende JDBC Datentypen lesen: LONGVARBINARY

getDate: DATE (empfohlene Methode)

kann auch folgende JDBC Datentypen lesen: CHAR, VARCHAR, LONGVARCHAR, TIMESTAMP

getTime: TIME (empfohlene Methode)

kann auch folgende JDBC Datentypen lesen: CHAR, VARCHAR, LONGVARCHAR, TIMESTAMP

getTimestamp: TIMESTAMP (empfohlene Methode)

kann auch folgende JDBC Datentypen lesen: CHAR, VARCHAR, LONGVARCHAR, DATE, TIME

getAsciiStream: LONGVARCHAR (empfohlene Methode)

kann auch folgende JDBC Datentypen lesen: CHAR, VARCHAR, BINARY, VARBINARY, LONGVARBINARY

getUnicodeStream: LONGVARCHAR (empfohlene Methode)

kann auch folgende JDBC Datentypen lesen: CHAR, VARCHAR, BINARY, VARBINARY, LONGVARBINARY

getBinaryStream: LONGVARBINARY (empfohlene Methode)

kann auch folgende JDBC Datentypen lesen: BINARY, VARBINARY

getObject: (keine empfohlene Methode)

kann auch folgende JDBC Datentypen lesen: TINYINT, SMALLINT, INTEGER, BIGINT, REAL, FLOAT, DOUBLE, DECIMAL, NUMERIC, BIT, CHAR, VARCHAR, LONGVARCHAR, BINARY, VARBINARY, LONGVARBINARY, DATE, TIME, TIMESTAMP

1.6. Mutieren / Updaten von Tabellen

Nach der ersten Geschäftswoche möchte der Coffee Shop seine Datenbank auf den aktuellen Stand bringen. Dazu muss die Spalte SALES mutiert werden.

Die SQL Anweisung dazu sieht folgendermassen aus:

```
String updateString = "UPDATE COFFEES " +
  "SET SALES = 75 " +
  "WHERE COF_NAME LIKE 'Colombian'";
```

Mit Hilfe des `Statement` Objekts `stmt` kann diese Anweisung folgendermassen in eine JDBC Anweisung umgewandelt werden:

```
stmt.executeUpdate(updateString);
```

Nach dem Ausführen des Programms sieht die Tabelle COFFEES nun folgendermassen aus:

COF_NAME	SUP_ID	PRICE	SALES	TOTAL
Colombian	101	7.99	75	0
French_Roast	49	8.99	0	0
Espresso	150	9.99	0	0
Colombian_Decaf	101	8.99	0	0
French_Roast_Decaf	49	9.99	0	0

Bisher haben wir die Spalte TOTAL noch nicht mutiert. Daher liefert eine Auswertung an dieser Stelle lauter Nullen.

Als Übung wollen wir eine kleine Auswertung programmieren:
die Abfrage soll das Ergebnis

```
In dieser Woche wurden <Anzahl verkaufter Pfunde Kaffee> Pfund Kaffee
der Sorte <Kaffeesorte> verkauft.
```

Die entsprechende SQL Anweisung sieht folgendermassen aus:

```
String query = "SELECT COF_NAME, SALES FROM COFFEES " +
         "WHERE COF_NAME LIKE 'Colombian'";
ResultSet rs = stmt.executeQuery(query);
while (rs.next()) {
          String s = rs.getString("COF_NAME");
          int n = rs.getInt("SALES");
          System.out.println("In dieser Woche wurden "+n +
                             " Pfund Kaffee der Sorte " + s +
                                           " verkauft.");
}
```

Diese Auswertung liefert die Ausgabe:

```
In dieser Woche wurden 75 Pfund Kaffee der Sorte Colombian verkauft.
```

Da die WHERE Klausel die Selektion einschränkt, kann nur eine einzelne Zeile im Resultset stehen. Wir können uns daher die WHILE Schleife sparen:

```
rs.next();
String s = rs.getString(1);
int n = rs.getInt(2);
System.out.println("In dieser Woche wurden "+n +
                    " Pfund Kaffee der Sorte " + s +
            " verkauft.");
```

`rs.next()` muss allerdings stehen, weil der Cursor nach der Abfrage *vor* der ersten Zeile im Resultset steht.

Mit all diesen Ergänzungen sieht unser Programm zum Mutieren der Tabelle COFFEES folgendermassen aus:

```
package updatecoffees;

import java.sql.*;

public class UpdateCoffees {

      public static void main(String args[]) {
          System.out.println("[UpdateTable]Start");
          String url = "jdbc:odbc:coffeebreak";
        Connection con;
          Statement stmt;

        try {
            System.out.println("Verbindung zum Driver aufbauen");
          Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        } catch(java.lang.ClassNotFoundException e) {
            System.err.print("ODBC : ClassNotFoundException: ");
            System.err.println(e.getMessage());
        }

        try {
            System.out.println("Verbindung zur Datenbank aufbauen");
          con = DriverManager.getConnection(url,"", "");
            System.out.println("SQL Statement vorbereiten");
          stmt = con.createStatement();

            /**
             * Mutieren der COFFEES Tabelle, Spalte SALES
             */
            System.out.println("Mutieren der Tabelle COFFEES");
            String updateString = "UPDATE COFFEES SET SALES = 75 "+
                                "WHERE COF_NAME LIKE 'Colombian'";
            stmt.executeUpdate(updateString);
            stmt.close();
            con.close();
        } catch(SQLException ex) {
          System.err.println("SQLException: " + ex.getMessage());
            System.err.flush();
        }
          /**
           * Anzeigen der mutierten Tabelle
           */


          try {
            System.out.println("Aktueller Stand der Kaffeeverkaeufe");
            String selectString = "SELECT * FROM COFFEES";
            System.out.println("Verbindung zur Datenbank aufbauen");
          con = DriverManager.getConnection(url,"", "");
            System.out.println("SQL Statement vorbereiten");
```

```
        stmt = con.createStatement();
          ResultSet rs = stmt.executeQuery(selectString);
          String strTab= "";
          System.out.println("COF_NAME\t\tSUP_ID\tPRICE\tSALES\tTOTAL");
          while (rs.next()) {
          String sCOF = rs.getString("COF_NAME");
          int iSUP = rs.getInt("SUP_ID");
          float fPRICE = rs.getFloat("PRICE");
          int iSALES = rs.getInt("SALES");
          int iTOTAL = rs.getInt("TOTAL");
          strTab= "\t";
          if (sCOF.length()<15) strTab="\t\t";
            System.out.println(sCOF + strTab +
                     iSUP+"\t"+fPRICE+"\t"+iSALES+"\t"+iTOTAL);
          }
          stmt.close();
          con.close();
      } catch(SQLException ex) {
        System.err.println("SQLException: " + ex.getMessage());
      }
        // Einfache Auswertung
        try {
          System.out.println("\n\n");
          String selectString = "SELECT COF_NAME, SALES FROM COFFEES " +
                                    "WHERE COF_NAME LIKE 'Colombian'";
        con = DriverManager.getConnection(url,"", "");
        stmt = con.createStatement();
          ResultSet rs = stmt.executeQuery(selectString);
          while (rs.next()) {
          String s = rs.getString("COF_NAME");//oder getString(1)
          int n = rs.getInt("SALES"); // oder getInt(2)
          System.out.println("In dieser Woche wurden "+n +
                             " Pfund Kaffee der Sorte " + s +
                                  " verkauft.");
          }
          System.out.println("\n");
          rs = stmt.executeQuery(selectString);
          rs.next();
          String s = rs.getString(1);
          int n = rs.getInt(2);
          System.out.println("In dieser Woche wurden "+n +
                 " Pfund Kaffee der Sorte " + s +
            " verkauft.");
          System.out.println("\n\n");
          stmt.close();
          con.close();
      } catch(SQLException ex) {
        System.err.println("SQLException: " + ex.getMessage());
      }
        System.out.println("[UpdateTable]Ende");
    }
}
```

1.6.1. Zusammenfassen der Grundlagen

Bisher haben wir uns mit den Grundlagen von JDBC beschäftigt. Sie lernten, wie man eine Tabelle kreiert, Werte einfügt, die Tabelle abfragt und mutiert. Dies sind die Grundfunktionen von JDBC, so wie sie seit JDBC 1.0, also seit der Einführung von Java, bekannt sind.

Probleme traten vermutlich kaum auf, weil wir keine native Treiber einsetzen, sondern lediglich mit JDBC-ODBC Bridge gearbeitet haben.

Der Nachteil dieser Bridge ist die Performance. Aber die spielte bisher kaum eine entscheidende Rolle.

In den folgenden Abschnitten werden wir uns mit weitergehenden Konzepten beschäftigen.

1.7. Einsatz von Prepared Statements

Oft ist es einfacher und effizienter ein `PreparedStatement` Objekt einzusetzen, um SQL Anweisungen an die Datenbank zu senden. Diese Klasse wird aus der Klasse `Statement` hergeleitet, die wir bereits kennen.

1.7.1. Wann sollte man PreparedStatement Objekte einsetzen?

Falls Sie ein `Statement` Objekt öfters ausführen wollen, können Sie normalerweise die Ausführungszeit mit Hilfe eines `PreparedStatement` verkützen.

Der Hauptunterschied zu normalen `Statement` Objekten besteht darin, dass Sie dem `PreparedStatement` eine SQL Anweisung mitgeben. Diese SQL Anweisung wird an die DBMS gesandt und übersetzt. Ein `PreparedStatement` Objekt enthält damit eine übersetzte SQL Anweisung. Dies führt zu einer Beschleunigung bei der Ausführung.

Ein PreparedStatement kann ohne oder mit Parameter verwendet werden. Ohne Parameter ist der Nutzen geringer, da in diesem Fall einfach eine Anweisung vorübersetzt wird und hoffentlich mehrfach ausgeführt wird, da sonst kein Nutzen entstehen würde.
Vorteile entstehen, sobald Sie die SQL Anweisung mit Parametern versehen und jeweils bei der Ausführung diesen Parametern Werte zuordnen. Wie dies geschieht, sehen Sie in den nächsten Abschnitten.

1.7.2. Kreieren eines `PreparedStatement` Objekts

Wie bei den `Statement` Objekten können Sie `PreparedStatement` Objekte mit einer `Connection` Methode kreieren. Die folgende Anweisung verwendet zwei Eingabe Parameter:

```
PreparedStatement updateSales = con.prepareStatement(
    "UPDATE COFFEES SET SALES = ? WHERE COF_NAME LIKE ?");
```

Die Variable `updateSales` enthält eine SQL Anweisung, "UPDATE COFFEES SET SALES = ? WHERE COF_NAME LIKE ?" , welche auch an die DBMS geschickt werden kann und dort, falls die DBMS dies ermöglicht, übersetzt wird.

1.7.3. Zuordnung von Werten zu den PreparedStatement Parametern

Den Fragezeichen in der obigen Anweisung müssen wir Werte zuweisen. Die Fragezeichen sind einfach Platzhalter. Falls Sie in einem `PreparedStatement` fehlen, brauchen wir natürlich auch keine Werte zuzuweisen.

Falls der Wert, den Sie der vorübersetzen Anweisung zuweisen wollen, vom Typ int ist, geschieht diese Zuweisung mit Hilfe der Methode `setInt()`, falls es sich um eine Zeichenkette handelt, ist die entsprechende Methode `setString()`. Ganz allgemein rufen Sie eine `setXXX()` Methode auf, um Werte einem `PreparedStatement` zuzuweisen.

Falls wir die Anweisung aus dem vorigen Abschnitt reproduzieren wollen, also die Verkaufszahlen mutieren wollen, wäre die erste Wertzuweiseung durch folgende Programmzeile gegeben (erster Parameter, int Wert 75):

```
updateSales.setInt(1, 75);
updateSales.setString(2, "Colombian");
```

Der erste Parameter erhält den Wert (int) 75, der zweite den Wert (String) Colombian.

Danach müssen wir das PreparedStatement ausführen. Die folgenden beiden Programmcodefragmente erledigen beide die selbe Aufgabe.

Variante 1:

```
String updateString = "UPDATE COFFEES SET SALES = 75 " +
                "WHERE COF_NAME LIKE 'Colombian'";
stmt.executeUpdate(updateString);
```

Variante 2:

```
PreparedStatement updateSales = con.prepareStatement(
  "UPDATE COFFEES SET SALES = ? WHERE COF_NAME LIKE ? ");
updateSales.setInt(1, 75);
updateSales.setString(2, "Colombian");
updateSales.executeUpdate();
```

Die Methode `executeUpdate()` wird in beiden Fällen eingesetzt. Aber im zweiten Fall, beim `PreparedStatement` wird kein Parameter benutzt. Die Werte wurden bereits vorher gesetzt.

Bisher hat die neue Methode keinerlei Vorteile. Aber sobald wir eine Anweisung mehrfach ausführen, kommen die Vorteile des `PreparedStatement` zum Zuge:

falls Sie beispielsweise öfters eine Mutation durchführen wollen oder müssen, gewinnen Sie beträchtlich an Ausführungszeit, indem Sie vorübersetzte Anweisungen verwenden.

Falls Sie einem Parameter einen Wert zuweisen, bleibt dieser Wert solange bestehen, bis er überschrieben oder gelöscht wird, mit `clearParameter()`. Das folgende Programmfragment zeigt dies an einem einfachen Beispiel:

```
updateSales.setInt(1, 100);
updateSales.setString(2, "French_Roast");
updateSales.executeUpdate();
// SALES von French Roast auf 100 setzen
updateSales.setString(2, "Espresso");
updateSales.executeUpdate();
// SALES von Espresso auf 100 setzen
// der erste Parameter bleibt unverändert
// der zweite Parameter wurde mutiert
// zu "Espresso"
```

1.7.4. Schleifen und PreparedStatements

Oft kann ein Mutations- oder Einfügungs-Programm vereinfacht werden, wenn Sie PreparedStatements in einer Schleife verwenden und dann in jeder Schleife die Werte aktualisieren, die Sie an das `PreparedStatement` Objekt übergeben.

Im folgenden Programmfragment wird eine Schleife eingesetzt, um die Werte in einem PreparedStatement zu setzen, welches die Verkaufszahlen aktualisiert. Im Array `salesForWeek` werden die wöchentlichen Verkaufszahlen abgespeichert:
der erste Wert (175) entspricht den Verkäufen der Marke "Colombian", der zweite Wert (150) der Kaffeesorte "French_Roast" und so weiter.

```
PreparedStatement updateSales;
String updateString = "update COFFEES " +
                "set SALES = ? where COF_NAME like ?";
updateSales = con.prepareStatement(updateString);
int [] salesForWeek = {175, 150, 60, 155, 90};
String [] coffees = {"Colombian", "French_Roast", "Espresso",
               "Colombian_Decaf", "French_Roast_Decaf"};
int len = coffees.length;
for(int i = 0; i < len; i++) {
          updateSales.setInt(1, salesForWeek[i]);
          updateSales.setString(2, coffees[i]);
          updateSales.executeUpdate();
}
```

Nun kann der Shop Besitzert die Tabelle viel schneller mutieren. Allerdings ist die Lösung unglücklich, weil alles hart einprogrammiert wurde. Aber das lässt sich leicht beheben, mittels eines GUIs.

1.7.5. Rückgabewerte der Method `executeUpdate()`

Während die Methode `executeQuery()` ein `ResultSet` Objekt zurückliefert, mit dem Ergebnis der Abfrage der DBMS, liefert `executeUpdate()` die Anzahl Zeilen, welche mutiert wurden:

```
updateSales.setInt(1, 50);
updateSales.setString(2, "Espresso");
int n = updateSales.executeUpdate();
// n = 1 weil sicher nur eine Zeile verändert wurde
```

Die Tabelle COFFEES wurde mutiert (50 wurde für die Kaffeesorte Espresso eingefügt). n ist somit 1. Falls Sie mit dieser Methode eine CREATE TABLE Anweisung auführen würden, würde der Rückgabewert 0 sein:

```
int n = executeUpdate(createTableCoffees); // n = 0
```

Ein Rückgabewert 0 bedeutet also zweierlei:
(1) die Anweisung, welche ausgeführt wurde, veränderte keine Zeile oder
(2) die Anweisung war eine DDL Anweisung.

Ein vollständiges Beispiel finden Sie auf der CD / dem Server.

1.8. Verknüpfen von Tabellen - Joins

Oft benötigen Sie mehrere Tabellen, um die Daten zusammenzustellen, welche Sie für eine bestimmte Auswertung benötigen.

Beispiel:
der Shop Inhaber möchte wissen, wieviel Kaffee er von der Firma Acme, Inc verkauft. Die Verkaufszahlen sind in der Tabelle COFFEES, die Händlerinformationen in der Tabelle SUPPLIERS. Beiden Tabellen gemeinsam ist die Spalte SUP_ID. Diese kommt also in beiden Tabellen vor.

Aber bevor wir diese Verknüpfung ausführen können, müssen wir die Tabelle SUPPLIERS kreieren und die Daten einfügen:

```
String createSUPPLIERS = "CREATE TABLE SUPPLIERS " +
                   "(SUP_ID INTEGER, SUP_NAME VARCHAR(40), " +
                   "STREET VARCHAR(40), CITY VARCHAR(20), " +
                   "STATE CHAR(2), ZIP CHAR(5))";
stmt.executeUpdate(createSUPPLIERS);
```

(das Programm befindet sich auf dem Server / der CD : das JBuilder Projekt CreateTable enthält die Programme zum Kreieren der in diesen Beispielen verwendeten Tabellen) beziehungsweise:

```
stmt.executeUpdate("insert into SUPPLIERS values (101, " +
    "'Acme, Inc.', '99 Market Street', 'Groundsville', " +
"'CA', '95199'");
stmt.executeUpdate("Insert into SUPPLIERS values (49," +
    "'Superior Coffee', '1 Party Place', 'Mendocino', 'CA', " +
"'95460'");
stmt.executeUpdate("Insert into SUPPLIERS values (150, " +
    "'The High Ground', '100 Coffee Lane', 'Meadows', 'CA', " +
"'93966'");
```

Die Tabelle fragen wir mit folgender SQL Anweisung ab:

```
ResultSet rs = stmt.executeQuery("select * from SUPPLIERS");
```

mit folgendem Ergebnis (ohne Strasse aus Platzgründen):

SUP_ID	SUP_NAME	CITY	STATE	ZIP
49	Superior Coffee	Mendocino	CA	95460
101	Acme, Inc.	Groundsville	CA	95199
150	The High Ground	Meadows	CA	93966

Nachdem wir alle Daten vorbereitet haben, können wir auch die Auswertung kreieren, die wir oben erwähnt hatten.

Da beide Tabellen, SUPPLIERS und COFFES, die Spalte SUP_ID haben, können wir damit eine Verknüpfung der beiden Tabellen, einen JOIN erreichen.

Die folgende Anweisung selektiert die Kaffeesorten, welche von Acme, Inc. gekauft werden:

```
String query = "
SELECT COFFEES.COF_NAME " +
   "FROM COFFEES, SUPPLIERS " +
   "WHERE SUPPLIERS.SUP_NAME LIKE 'Acme, Inc.' " +
   "and SUPPLIERS.SUP_ID = COFFEES.SUP_ID";

ResultSet rs = stmt.executeQuery(query);
System.out.println("Coffees bought from Acme, Inc.: ");
while (rs.next()) {
    String coffeeName = rs.getString("COF_NAME");
    System.out.println("     " + coffeeName);
}
```

Falls Sie das Join Beispiel auf der CD / dem Server starten, erhalten Sie diese Auswertung:

```
Supplier          Coffee:
Acme, Inc.        Colombian
Acme, Inc.        Colombian_Decaf
```

1.9. Transaktionen

In einigen Anwendungen sind Sie gezwungen mehrere Aktionen entweder als Ganzes oder überhaupt nicht auszuführen. Ein anderer typischer Fall sind Systeme, bei denen eine Aktion lediglich ausgeführt werden soll, nachdem eine erste erfolgreich abgeschlossen wurde.

In unserem Beispiel möchten wir das Verkaufstotal pro Tag und Woche erfassen. Aber es macht wenig Sinn die eine Zahl in die Tabelle einzutragen, wenn die andere nicht gespeichert werden kann.

Solche Fälle kann man mit einer Transaktion abdecken. Eine Transaktion besteht aus einer oder mehreren Anweisungen, die zusammen ausgeführt werden sollten, also entweder ganz oder überhaupt nicht!

1.9.1. Ausschalten des Auto-commit Modus

Falls Sie eine Verbindung zur Datenbank herstellen, ein `Connection` Objekt bestimmen, dann geschieht dies in der Regel im Auto-Commit Modus. Das heisst, dass jedes individuelle SQL Statement als Transaktion aufgefasst wird und sofort bestätigt wird, gleich nach dessen Ausführung.

Eine Anweisung ist vollständig, wenn alle ResultSets und Update Zähler vorliegen. In den meisten Fällen wird eine Anweisung gleich nach deren Ausführung bestätigt, also committed.

Sie können mehrere Anweisungen zu einer Transaktion zusammenfassen, indem Sie Autocommit einfach ausschalten und diese Aufgabe selber übernehmen. Dies geschieht, indem Sie `AutoCommit` auf `false` setzen:

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

1.9.2. Committen einer Transaktion

Nachdem Sie den Auto-commit Modus ausgeschaltet haben, wird keine SQL Anweisung mehr bestätigt, bis Sie dies explizit verlangen.

Alle Anweisungen seit dem letzten Commitment sind Teil dieser Transaktion. Sie bestätigen alle Anweisungen seit dem letzten Commit. Das folgende Beispiel zeigt wie dies aussehen könnte:

```
con.setAutoCommit(false);
PreparedStatement updateSales = con.prepareStatement(
    "UPDATE COFFEES SET SALES = ? WHERE COF_NAME LIKE ?");
updateSales.setInt(1, 50);
updateSales.setString(2, "Colombian");
updateSales.executeUpdate();
PreparedStatement updateTotal = con.prepareStatement(
 "UPDATE COFFEES SET TOTAL = TOTAL + ? WHERE COF_NAME LIKE ?");
updateTotal.setInt(1, 50);
updateTotal.setString(2, "Colombian");
updateTotal.executeUpdate();
con.commit();
con.setAutoCommit(true);
```

In diesem Beispiel wird der auto-commit Modus ausgeschaltet. Somit werden die zwei SQL Anweisungen `updateSales` und `updateTotal` zu einer Transaktion zusammengefasst.

Wann immer die `commit()` Methode implizit (im auto-commit Modus) oder explizit ausgeführt wird, werden die Änderungen aus den Anweisungen bestätigt und permanent.

In unserem Fall wird das Verkaufstotal (SALES und TOTAL zur Kaffeesorte 'Columbian') von 0 auf 50 verändert.

Im Beispiel auf der CD / dem Server sehen Sie ein vollständiges Programm, welches hier der Vollständigkeit halber wiedergegeben wird:

```
package transaktionen;

import java.sql.*;

public class Transaktionen {

  public static void main(String args[]) {
    String url = "jdbc:odbc:coffeebreak";
    Connection con = null;
    Statement stmt;
    PreparedStatement updateSales;
    PreparedStatement updateTotal;
    String updateString = "update COFFEES " +
                          "set SALES = ? where COF_NAME like ?";
    String updateStatement = "update COFFEES " +
                       "set TOTAL = TOTAL + ? where COF_NAME like ?";
    String query = "select COF_NAME, SALES, TOTAL from COFFEES";

    try {
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    } catch(java.lang.ClassNotFoundException e) {
      System.err.print("ClassNotFoundException: ");
      System.err.println(e.getMessage());
    }
    try {
```

```
      con = DriverManager.getConnection(url, "myLogin", "myPassword");
      updateSales = con.prepareStatement(updateString);
      updateTotal = con.prepareStatement(updateStatement);
      int [] salesForWeek = {175, 150, 60, 155, 90};
      String [] coffees = {"Colombian", "French_Roast",
                  "Espresso", "Colombian_Decaf",
                        "French_Roast_Decaf"};
      int len = coffees.length;
      con.setAutoCommit(false);
      for (int i = 0; i < len; i++) {
        updateSales.setInt(1, salesForWeek[i]);
        updateSales.setString(2, coffees[i]);
        updateSales.executeUpdate();
        updateTotal.setInt(1, salesForWeek[i]);
        updateTotal.setString(2, coffees[i]);
        updateTotal.executeUpdate();
        con.commit();
      }
      con.setAutoCommit(true);
      updateSales.close();
      updateTotal.close();
      stmt = con.createStatement();
      ResultSet rs = stmt.executeQuery(query);
      while (rs.next()) {
        String c = rs.getString("COF_NAME");
        int s = rs.getInt("SALES");
        int t = rs.getInt("TOTAL");
        System.out.println(c + "     " +  s + "    " + t);
      }
      stmt.close();
      con.close();
    } catch(SQLException ex) {
      System.err.println("SQLException: " + ex.getMessage());
      if (con != null) {
        try {
          System.err.print("Die Transaktion wird rueckgaengig gemacht ");
          con.rollback();
        } catch(SQLException excep) {
          System.err.print("SQLException: ");
        System.err.println(excep.getMessage());
        }
      }
    }
  }
}
```

Wie im Beispiel oben, sollte man sich angewöhnen, nach der Transaktion den Commit Modus wieder auf Auto zu setzen. Damit vermeiden Sie auch Zugriffskonflikte und Sperren der Datenbank, die im schlimmsten Fall bei komplexen Transaktionen resultieren können.

1.9.3. Erhalten der Datenintegrität mittels Transaktionen

Zusätzlich zum Zusammenfassen mehrerer Anweisungen zu einer Einheit kann man Transaktionen auch einsetzen, um die Integrität der Daten in einer Tabelle zu gewährleisten.

Beispiel:
nehmen wir an, ein Mitarbeiter des Caffee Shops ist damit beauftragt neue Preise einzugeben. Aber aus irgend einem Grund verzögert sich diese Arbeit. In der Zwischenzeit sind die Kaffeepreise gestiegen und der Besitzer möchte nun die neuen Daten eingeben. Der Mitarbeiter und der Besitzer des Shops geben beide die Daten neu ein, wobei jene des Mitarbeiters bereits wieder veraltet sind.

Nach der Eingabe realisiert der Mitarbeiter, dass die Daten veraltet sind, die Preise bereits erhöht wurden. Er kann nun, falls das seine Anwendung erlaubt, die Transaktion rückgängig

machen, mit der `rollback()` Methode. Diese Methode bricht die Transaktion ab und setzt die Werte wieder auf die vor der Transaktion.

Zur selben Zeit druckt der Besitzer mit einer SELECT Anweisung die neuen Preise aus. Im schlimmsten Fall könnte es passieren, dass er die alten Preise ausdruckt.

Falls man Transaktionen einsetzt, kann man diese Problem-Situationen in den Griff bekommen. Zumindest kann eine Transaktion einen bestimmten Schutz bieten vor Zugriffskonflikten und inkonsistenten Daten.

Um Konflikte zu vermeiden, wird die DBMS Sperren verwenden, Locks. Damit wird der Zugriff auf Daten im kritischen Bereich durch mehrere Benutzer vermieden bzw. verunmöglicht.

Im Auto-commit Modus werden die Sperren nach jeder Anweisung aufgehoben.
Sonst, bei Transaktionen, bleibt eine Sperre solange bestehen, bis die Transaktion bestätigt wird.

Falls Sie Daten lesen, die später als ungültig aus der DBMS eliminiert werden (durch ein `rollback()`) dann spricht man von einem "dirty read", weil die Daten ungültig sind.

Wie Locks gesetzt werden, hängt von dem DBMS ab, genauer vom Transaktions Isolationslevel. Dieser kann von 'wird nicht unterstützt' bis zu sehr ausgefeilten Techniken auf Feldebene reichen.

Ein Beispiel für einen Isolationslevel ist TRANSACTION_READ_COMMITTED , welcher besagt, dass Daten erst gelesen werden dürfen, nachdem sie committed wurden.
Falls also in einer Transaktion der Islationslevel auf TRANSACTION_READ_COMMITTED gesetzt wird, verhindert die DBMS das Lesen von veralteten Daten, 'dirty reads'. Im Interface `Connection` sind fünf unterschiedliche Werte vordefiniert, mit denen der Transaktions-Isolationslevel in JDBC gesetzt werden kann.

Normalerweise brauchen Sie sich nicht um den Isolationslevel zu kümmern. Die DBMS verwendet einen Standardlevel, der in der Regel völlig ausreichend ist. Sie können mit einem Programm leicht den Isolationslevel Ihrer DBMS herausfinden. Sie finden auf der CD / dem Server ein entsprechende Programm: `getTransactionIsolation()`.

Falls Sie den Transaktions-Isolationslevel ändern wollen, können Sie dies mittels `Connection.setTransactionIsolation()`, wobei der Methodenaufruf nicht sbesagt: falls die DBMS das Setzen nicht unterstützt, geschieht nicht.

1.9.4. Wann sollte die `rollback()` Methode aufgerufen werden?

Wie bereits erwähnt, bricht die rollback() Methode eine Transaktion ab. Alle veränderten Datenfelder werden zurückgesetzt. Sie können Sie Transaktion erneut ausführen. Die `rollback()` Methode sollten Sie immer dann einsetzen, wenn beim Ausführen einer Transaktion eine `SQLException` auftritt. In diesem Fall wissen Sie ja nicht genau, was wie vollständig ausgeführt wurde.

Das folgende Beispiel zeigt, wie dies aussehen könnte (selbes Beispiel wie oben):

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

```
    for (int i = 0; i < len; i++) {
      updateSales.setInt(1, salesForWeek[i]);
      updateSales.setString(2, coffees[i]);
      updateSales.executeUpdate();
      updateTotal.setInt(1, salesForWeek[i]);
      updateTotal.setString(2, coffees[i]);
      updateTotal.executeUpdate();
      con.commit();
    }
    con.setAutoCommit(true);
    ...
    con.close();
  } catch(SQLException ex) {
    System.err.println("SQLException: " + ex.getMessage());
    if (con != null) {
      try {
        System.err.print("Die Transaktion wird rueckgaengig gemacht ");
        con.rollback();
      } catch(SQLException excep) {
        System.err.print("SQLException: ");
      System.err.println(excep.getMessage()); ...
```

1.10. Stored Procedures

Eine 'Stored Procedure' besteht aus einer Gruppe von SQL Anweisungen, die zusammen eine logische Einheit bilden, oder aus Abfragen, welche auf dem Server gespeichert werden.

Beispielsweise könnten einfache Auswertungen vorprogrammiert werden und auf der Datenbank abgespeichert werden. Dann kann (fast) jeder sie einfach aufrufen, wo immer er ist, sofern er Zugriff auf den DBMS Server hat.

Solche Prozeduren können IN, OUT und INOUT Parameter haben, also Eingabe-, Ausgabe und Ein/Ausgabe- Parameter.

Diese Stored Procedures werden allerdings nicht von allen DBMS unterstützt, beispielsweise nicht von Excel, Textdatein, Access und vielen mehr.

Daher werden wir im folgenden Abschnitt einfach zeigen, wie solche Programme aussehen könnten, ohne sie allerdings ausführen zu können.

1.10.1. SQL Statements zum Kreieren einer Stored Procedure

In diesem Abschnitt schauen wir uns Stored Procedures an, welche keine Parameter haben. Das Beispiel dient einfach der Illsutration der Konzepte, also weniger als produktives Beispiel.

Die Syntax der in der DBMS gespeicherten Prozeduren unterscheidet sich von DBMS zu DBMS. Einige benutzen `begin ... end`, andere Klammern und weitere Schlüsselwörter. Hier ein Beispiel für eine fiktive DBMS:

```
create procedure SHOW_SUPPLIERS
as
select SUPPLIERS.SUP_NAME, COFFEES.COF_NAME
from SUPPLIERS, COFFEES
where SUPPLIERS.SUP_ID = COFFEES.SUP_ID
order by SUP_NAME
```

Diese Anweisung würden wir wie gehabt in eine Zeichenkettenvariable stecken und später in einem Methodenaufruf verwenden.

```
String createProcedure = "create procedure SHOW_SUPPLIERS " +
          "as " +
          "select SUPPLIERS.SUP_NAME, COFFEES.COF_NAME"+
          "from SUPPLIERS, COFFEES " +
           "where SUPPLIERS.SUP_ID = COFFEES.SUP_ID " +
           "order by SUP_NAME";
```

Mit dem Connection Objekt würde diese Prozedur an die DBMS übergeben:

```
Statement stmt = con.createStatement();
stmt.executeUpdate(createProcedure);
```

Auf der Serverseite wird diese Prozedur übersetzt und in der DBMS gespeichert.

1.10.2. Aufruf einer gespeicherten Procedure aus JDBC

JDBC gestattet den Aufruf einer gespeicherten Prozedur mittels eines `CallableStatement` Objekts. Genau wie bei den `Statement` und `PreparedStatement` Objekten werden `CallableStatement` Objekte durch offene `Connection` Objekte kreiert.

Ein `CallableStatement` enthält einen Aufruf einer Stored Procedure, nicht die Stored Procedure selbst. Im Programmfragment unten wird zuerst ein Aufruf der Stored Procedure SHOW_SUPPLIERS mittels der Connection con durchgeführt.

Der Treiber muss die Escape Syntax "{call SHOW_SUPPLIERS}" umsetzen, in SQL Anweisungen bzw. Aufrufe der Stored Procedure.

```
CallableStatement cs = con.prepareCall("{call
SHOW_SUPPLIERS}");
ResultSet rs = cs.executeQuery();
```

Das Resultset könnte dann beispielsweise folgendermassen aussehen:

```
SUP_NAME                  COF_NAME
----------------          ------------------------
Acme, Inc.                        Colombian
Acme, Inc.                        Colombian_Decaf
Superior Coffee                   French_Roast
Superior Coffee                   French_Roast_Decaf
The High Ground                   Espresso
```

Der Aufruf von `cs.exeuteQuery()` geschieht, weil cs eine Stored Procedure aufruft, welche eine Abfrage, ein Query, enthält.

Falls in der SQL Anweisung eine DDL Anweisung stehen würde, müsste `executeUpdate()` aufgerufen werden.

Falls die Prozedur mehr als eine Abfrage oder mehr als ein DDL enthält, werden mehrere ResultSets und mehrere Update Zähler zurückgeliefert.

Die Klasse `CallableStatement` ist eine Unterklasse der Klasse `PreparedStatement`. Jedes `CallableStatement` kann also Eingabeparameter IN genau so verarbeiten, wie `PreparedStatement`. Genau so kann ein `CallableStatement` auch Ausgabeparameter OUT enthalten oder INOUT Parameter.

1.11. Kreieren einer vollständigen JDBC Applikation

Bisher haben wir viele kleine Programme und Programmfragmente gesehen. Schön wäre es, eine vollständige Applikation zu haben, mit GUI und Menüs und allen tollen Sachen.

Auf der CD / dem Server finden Sie Programme zum Kreieren der Tabellen (beider Tabellen: im entsprechenden Projekt sind zwei Java Programme), zum Einfügen der Daten und für verschiedene Abfragen sowie `CallableStatements`.

Verschiedene Programme zeigen Ihnen, beispielsweise welchen Isolationslevel eine bestimmte ODBC Datenquelle besitzt. Der dortige Hilfstext stammt direkt aus der JDK Dokumentation.

1.11.1. Import von Klassen

In den Beispielen verwenden wir JDBC, also müssen wir auch die JDBC Basisklassen bzw. die Interfacebeschreibungen importieren:

```
import java.sql.*;
```

Falls Sie weitergehende Hilfsmittel einsetzen, wie beispielsweise GUI Elemente oder Hilfsklassen, müssen Sie die Imports entsprechend ergänzen.

Es gibt auch eine Erweiterung der JDBC Basisklassen. Diese befinden sich im Package:

```
javax.sql
```

Das x zeigt an, dass es sich um Erweiterungen des Java Standards handelt. Darin werden insbesondere die neueren JDBC Datentypen (CLOB, BLOB, ...) unterstützt, wobei auch hier gilt: sofern die DBMS dazu in der Lage ist.

SQLException liefert Ihnen insgesamt drei Bestandteile:
1) die Message
2) den SQL Zustand (SQL State), der den Zustand / den Fehler gemäss den X/Open SQLState Conventions anzeigt,
3) und dem Vendor Error Code (einer Zahl, die der Anbieter des Treibers beschreiben muss)

Sie können auf diese drei Komponenten einzeln zugreifen:
1) getMessage ,
2) getSQLState und
3) getErrorCode .

```
try {
  // irgend ein Programmcode
} catch(SQLException ex) {
  System.out.println("\n--- SQLException ---\n");
  while (ex != null) {
          System.out.println("Message:   "+ ex.getMessage ());
          System.out.println("SQLState:  "+ ex.getSQLState ());
          System.out.println("ErrorCode: "+ ex.getErrorCode());
```

```
            ex = ex.getNextException();
            System.out.println("");
    }}
```

Beispielsweise falls Sie die Tabellen zweimal kreieren wollten:

```
--- SQLException ---
Message:  There is already an object named 'COFFEES' in the
database.
Severity 16, State 1, Line 1
SQLState: 42501
ErrorCode:   2714
```

SQLState wird in X/Open und ANSI-92 definiert.
Hier zwei Beispiele:

```
08001 -- No suitable driver
HY011 -- Operation invalid at this time
```

1.11.2. Das Behandeln von Warnungen

Warnungen sind eine Unterklasse der SQLExceptions. Sie zeigen beispielsweise an, dass eine Verbindung unterbrochen wurde.

Hier ein Beispiel , wie eine Warnung ausgewertet werden könnte:

```
Statement stmt = con.createStatement();
ResultSet rs=stmt.executeQuery("select COF_NAME from COFFEES");
while (rs.next()) {
  String coffeeName = rs.getString("COF_NAME");
  System.out.println("Coffees available at the Coffee Break:");
  System.out.println("    " + coffeeName);
  SQLWarning warning = stmt.getWarnings();
  if (warning != null) {
          System.out.println("\n---Warnung---\n");
          while (warning != null) {
                  System.out.println("Message: "
                                + warning.getMessage());
                  System.out.println("SQLState: "
                                + warning.getSQLState());
                  System.out.print("Vendor error code: ");
                  System.out.println(warning.getErrorCode());
                  System.out.println("");
                  warning = warning.getNextWarning();
          }
  }
  SQLWarning warn = rs.getWarnings();
  if (warn != null) {
          System.out.println("\n---Warnung---\n");
          while (warn != null) {
                  System.out.println("Message: "
                                + warn.getMessage());
                  System.out.println("SQLState: "
                                + warn.getSQLState());
                  System.out.print("Vendor error code: ");
                  System.out.println(warn.getErrorCode());
                  System.out.println("");
                  warn = warn.getNextWarning();
          }
  }
}
```

1.12. Starten der Beispiele

Die Programme wurden alle mit JBuilder 4 erstellt. Sie können jederzeit direkt aus einem DOS Fenster heraus die Programme starten.

Ein typischer Ablauf wäre:

1) ins Verzeichnis über der Class Datei wechseln
2) Aufruf der JVM
   java -cp .;.. <packageName>.<Name der ClassDate ohne .class>

Alle Programme sollten funktionieren, sofern die ODBC Datenquellen korrekt aufgesetzt sind.

Starten Sie mit dem einfachst möglichen Programm: demVerbindungsaufbau und sonst nichts!

Wenn dieser nicht klappt, können Sie eh alles andere vergessen.

1.13. Applet Abfrage der Daten

Um das Abfragen auch über das Internet zu gestatten, wurde ein einfaches Applet entwickelt, welches auf die ODBC Datenquellen zugreifen kann.

Das Erstellen des Applets ist denkbar einfach:

1.13.1. Das Applet

Applets haben ihre Tücken, speziell falls Sie ein Package verwenden. Daher der Ratschlag: keine Packages bei Applets!

Hier einige generelle Ratschläge zum Einsatz von Applets für Datenbankabfragen:

1) kapseln Sie den gesamten JDBC Programmcode in einen separaten Thread
2) Statusmeldungen, wie beispielsweise der verzögerte Aufbau einer grafischen Darstellung, müssen angezeigt werden. Auch falls der Verbindungsaufbau zur DBMS lange dauert muss eine Statusmeldung generiert werden.
3) Fehlermeldungen müssen auf dem Bildschirm angezeigt werden, also nicht in `System.out` oder `System.err`.

1.13.2. Der Applet Programmcode

Da ich es nicht lassen konnte das Applet im JBuilder zu entwickeln, enthält die folgende Version ein Package. Beim Ausführen gibt es daher Probleme. Um diese zu umgehen, steht eine Batch Datei zur Verfügung, mit der das Applet korrekt gestartet werden kann.

```
package ausgabeapplet;

import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import java.applet.Applet;
import java.awt.Graphics;
import java.util.Vector;
import java.sql.*;


public class JDBCAusgabeApplet extends Applet implements Runnable {
```

```
private Thread worker;
private Vector queryResults;
private String message = "Initialisierung";

public synchronized void start() {
  // mit start wird eine neue Abfrage gestartet
  // also die DB abgefragt
  if (worker == null) {
      message = "Verbindungsaufbau zur Datenbank";
        worker = new Thread(this);
      worker.start();
  }
}

/**
 * run startet den Worker Thread
 * Dieser fragt evtl die DB ab und das kann dauern
 * Aber es tritt kein Zugriffskonflikt auf / kein sync.
 */

public void run() {
  String url = "jdbc:odbc:coffeebreak";
  String query = "select COF_NAME, PRICE from COFFEES";

  try {
      Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");
  } catch(Exception ex) {
      setError("Laden des Datenbank Treibers schlug fehl: " + ex);
      return;
  }

  try {
      Vector results = new Vector();
      Connection con = DriverManager.getConnection(url,"", "");
      Statement stmt = con.createStatement();
      ResultSet rs = stmt.executeQuery(query);
      while (rs.next()) {
        String s = rs.getString("COF_NAME");
        float f = rs.getFloat("PRICE");
        String text = s + "     " + f;
        results.addElement(text);
      }
      stmt.close();
      con.close();
      setResults(results);

  } catch(SQLException ex) {
      setError("SQLException: " + ex);
  }
}
```

```
    /**
     * paint muss das Ergebnis anzeigen
     * hoffentlich
     */
    public synchronized void paint(Graphics g) {
      // liegt ein Ergebnis vor?
      if (queryResults == null) {
          g.drawString(message, 5, 50);
          return;
      }
      // Display the results.
        g.drawString("",5,10);
      g.drawString("Kaffee Preis pro Pfund:  ", 15, 20);
      int y = 30;
      java.util.Enumeration enum = queryResults.elements();
      while (enum.hasMoreElements()) {
          String text = (String)enum.nextElement();
          g.drawString(text, 15, y+15);
          y = y + 15;
      }
    }
    /**
     * zwischenspeichern einer Fehlermeldung
     * Diese wird später angezeigt.
     */
    private synchronized void setError(String mess) {
      queryResults = null;
      message = mess;
      worker = null;
      // Anzeige
      repaint();
    }
    /**
     * Ergebnisse der Abfrage abspeichern
     * Angezeigt wird später
     */
    private synchronized void setResults(Vector results) {
      queryResults = results;
      worker = null;
      // AWT soll anzeigen
      repaint();
    }
}
```

1.13.3. Starten des Applets

Da wir die ODBC-JDBC Bridge verwenden, ist das Applet zwar im Intranet brauchbar, aber kaum über das Internet, weil die Clients speziell konfiguriert werden müssen. Falls Sie einen reinen JDBC Treiber zur Verfügung haben, ist dieses Problem behoben.

Zum Starten sollten Sie die folgende Batch Datei verwenden, Sie wurde im JBuilder generiert, weitestgehend:

```
@echo off
Set classpath= .;..
%JAVA_HOME%\bin\javaw -
Djava.security.policy="C:/WINNT/Profiles/zajoller.000/.jbuilder4/appletview
er.policy" sun.applet.AppletViewer JDBCAusgabeApplet.html
```

1.14. Das JDBC 2.0 API

In JDK ist das `java.sql` Package enthalten., auch unter JDBC 2.0 bekannt. Das JDK1.1 enthielt eine vereinfachte Version, JDBC 1.0. Wir haben also noch keinerlei Features benutzt, die neu hinzukamen, geschweige jene, die im Extension Package enthalten sind (`javax.sql`). JDBC 2.0 gestattet Ihnen:

- vorwärts und rückwärts zu scrollen, oder den Cursor zu einer bestimmten Zeile im ResultSet zu bewegen.
- Mutationen in der Datenbank mit Java Anweisungen, an Stelle von SQL Anweisungen im JDBC 1.0, durchzuführen.
- mehrere SQL Anweisungen als Einheit, im Batch an die Datenbank zu senden.
- die neuen SQL 3.0 Datentypen als Spaltenwerte einzusetzen.

1.14.1. Aufsetzen von JDBC 2.0

Damit Sie JDBC 2.0 benutzen können, müssen Sie

- JDK 1.2 oder neuer installieren und
- einen JDBC Treiber verwenden, der JDBC 2.0 unterstützt und schliesslich
- auf eine DBMS zugreifen, welche JDBC 2.0 Features implementiert

1.14.2. Bewegen des Cursors in Scrollable Result Sets

Eine der neuen Möglichkeiten im JDBC 2.0 API ist, den Cursor an eine bestimmte Stelle im Result Set zu bewegen und sowohl vorwärts, als auch rückwärts zu scrollen. Scrollable ResultSets gestatten es Ihnen leichter schicke GUIs zu konstruieren, in denen Sie vorwärts und rückwärts scrollen können, im ResultSet, nicht in der DBMS! Zudem können Sie jetzt auch direkt im ResultSet einzelne Zeilen mutieren, sofern Sie die ResultSets passend definieren:

```
Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                                     ResultSet.CONCUR_READ_ONLY);
ResultSet srs = stmt.executeQuery("SELECT COF_NAME, PRICE FROM
COFFEES");
```

Auf den ersten Blick sieht diese Anweisung ähnlich aus, wie gehabt. Aber bei der Definition des ResultSets verwenden wir zusätzliche Konstanten um den ResultSet Typus anzugeben:

- TYPE_FORWARD_ONLY ,
- TYPE_SCROLL_INSENSITIVE und
- **TYPE_SCROLL_SENSITIVE** .

Das zweite Argument beschreibt, ob das ResultSet mutierbar sein soll oder nicht:

- **CONCUR_READ_ONLY** und
- CONCUR_UPDATABLE .

Die beiden Parameter müssen gleichzeitig gesetzt werden: falls Sie angeben, ob ein ResultSet scrollbar sein soll, müssen Sie auch angeben, ob das ResultSet mutierbar sein soll.

ACHTUNG:
Der Typus muss zuerst angeben werden! Da beide Parameter int sind, würde der Compiler nicht merken, dass Sie die Parameter vertauscht haben.

1.14.2.1. TYPE_FORWARD_ONLY

Mit dieser Konstanten geben Sie an, dass sich der Cursor im ResultSet nur vorwärts bewegen kann. Falls Sie keinen Parameter angeben, ist dies der Standardwert.

1.14.2.2. TYPE_FORWARD_ONLY und CONCUR_READ_ONLY

Damit erhalten Sie ein ResultSet, welches scrollable ist und zwar von einem der Typen:

- TYPE_SCROLL_INSENSITIVE oder
- TYPE_SCROLL_SENSITIVE .

Die Differenz besteht darin, dass im ersten Fall Änderungen in der Datenbanken während der Lebensdauer des ResultSets nicht an diesen weitergegeben werden (der ResultSet merkt nichts von Mutationen in der Datenbank, er ist nicht sensitiv darauf).

Im zweiten Fall werden Änderungen, die in der Datenbank geschehen auch an das ResultSet weiteregegeben. Falls also zwei Benutzer auf die selben Daten zugreifen, der eine eine Auswertung in einem ResultSet ansieht, der andere abedr genau die dort vorhandenen Daten in er Datenbank mutiert, dann wird das ResultSet aktuell gehalten. Der Auswerter sieht also die Änderungen sofort. All dies gilt nur unter der Einschränkung: "... sofern die DBMS diese Features unterstützt".

1.14.3. Navigieren im Scrollable ResultSet

Nachdem Sie ein Scrollable ResultSet kreiert haben, können Sie den Cursor vorwärts, rückwärts oder gezielt zu einer Position bewegen. Standardmässig wird der Cursor vor die erste Zeile im ResultSet gestellt:

Vorwärts scrollen:

```
Statement stmt =
con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,

ResultSet.CONCUR_READ_ONLY);
ResultSet srs = stmt.executeQuery(
    "SELECT COF_NAME, PRICE FROM COFFEES");
while (srs.next()) {
  String name = srs.getString("COF_NAME");
  float price = srs.getFloat("PRICE");
  System.out.println(name + "     " + price);
}
```

mit folgendem Resultat:

```
Colombian         7.99
French_Roast      8.99
Espresso          9.99
Colombian_Decaf   8.99
French_Roast_Decaf9.99
```

Genau so gut können Sie das ResultSet rückwärts abfragen:

```
Statement stmt =
con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
```

```
ResultSet.CONCUR_READ_ONLY);
ResultSet srs = stmt.executeQuery("SELECT COF_NAME, PRICE FROM
COFFEES");
srs.afterLast();
while (srs.previous()) {
  String name = srs.getString("COF_NAME");
  float price = srs.getFloat("PRICE");
  System.out.println(name + "     " + price);
}
```

mit folgendem Ergebnis:

```
French_Roast_Decaf        9.99
Colombian_Decaf           8.99
Espresso                  9.99
French_Roast              8.99
Colombian                 7.99
```

Das Ergebnis ist das selbe, bis auf die Reihenfolge.

Sie können den Cursor aber auch absolut positionieren:

```
srs.absolute(4);
```

und ab dort vorwärts und rückwärts bewegen:

```
srs.absolute(4); // vierte Zeile
. . .
srs.relative(-3); // erste Zeile
. . .
srs.relative(2); // dritte Zeile
```

Es kann aber sein, dass der Treiber nur einen Teil dieser Methoden unterstützt. Sie sehen dies am Beispiel auf dem Server / der CD.

Mit vier weiteren Methoden können Sie die Position des Cursors im ResultSet verifizieren: isFirst , isLast , isBeforeFirst , isAfterLast .

Diese Methoden können Sie in bedingten Abfragen einsetzen. Hier ein Beispiel:

```
if (srs.isAfterLast() == false) {
  srs.afterLast();
}
while (srs.previous()) {
  String name = srs.getString("COF_NAME");
  float price = srs.getFloat("PRICE");
  System.out.println(name + "     " + price);
}
```

1.14.4. Mutationen des Result Sets

Falls der Treiber dies unterstützt, können Sie das ResultSet direkt mutieren und die Änderungen werden anschliessend vom System in der DBMS nachgeführt.

Bei der JDBC-ODBC Bridge mit Access funktioniert dies weitestgehend *nicht*!

Hier trotzdem ein kurzer Überblick über die Methoden, die dazu verwendet werden:

```
Connection con =
DriverManager.getConnection("jdbc:mySubprotocol:mySubName");
Statement stmt =
con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                            ResultSet.CONCUR_UPDATABLE);
ResultSet uprs = stmt.executeQuery("SELECT COF_NAME, PRICE FROM
COFFEES");
```

Das Ergebnis ist die übliche Tabelle. Mit der folgenden Anweisung würde das ResultSet, nicht direkt die DBMS mutiert, wegen den oben gesetzten Parametern!

```
stmt.executeUpdate("UPDATE COFFEES SET PRICE = 10.99" +
             "WHERE COF_NAME = FRENCH_ROAST_DECAF");
```

Analog könnten Sie auch eine einzelne Zeile mutieren:

```
uprs.last();
uprs.updateFloat("PRICE", 10.99f);
```

Die `updateXXX()` Methoden haben zwei Parameter: die Bezeichnung der Spalte und den Spaltenwert. Das Muster ist also völlig analog zu den `getXXX()` Methoden.

```
uprs.last();
uprs.updateFloat("PRICE", 10.99f);
uprs.updateRow();
```

Aus irgend einem Grund funktioniert die exakte Positionierung in Access nicht richtig. Daher steht der Cursor nach dem ersten Positionieren zwar an der korrekt Stelle, die relative Bewegung stimmt aber nicht.

Die Update Methoden sind so, dass Sie Änderungen auch rückgängig machen können:

```
uprs.last();
uprs.updateFloat("PRICE", 10.99);
uprs.cancelRowUpdates();
uprs.updateFloat("PRICE", 10.79);
uprs.updateRow();
```

1.14.5. Einfügen und Löschen einzelner Zeilen

Wir kennen bereits die Methode einzelne Zeilen in Tabellen einzufügen. Diese haben wir verwendet, um die Tabellen zu füllen:

```
stmt.executeUpdate("INSERT INTO COFFEES " +
                   "VALUES ('Kona', 150, 10.99, 0, 0)");
```

Mit Scrollable ResultSets kann man dies etwas anders auch bewältigen.
Als erstes muss man zur "Insert Row" Zeile wechseln, also der Position bei der die Zeile eingefügt werden soll:

```
moveToInsertRow()
```

Dann müssen Sie die Werte für die einzelnen Spalten definieren :

```
updateXXX()
```

Dann rufen Sie die

```
insertRow()
```

Methode auf, um die Werte einzufügen. De Werte werden simultan in den ResultSet und die Tabelle eingefügt.

```
Connection con =
DriverManager.getConnection("jdbc:mySubprotocol:mySubName");
Statement stmt =
con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                                ResultSet.CONCUR_UPDATABLE);
ResultSet uprs = stmt.executeQuery("SELECT * FROM COFFEES");

uprs.moveToInsertRow();
uprs.updateString("COF_NAME", "Kona");
uprs.updateInt("SUP_ID", 150);
uprs.updateFloat("PRICE", 10.99);
uprs.updateInt("SALES", 0);
uprs.updateInt("TOTAL", 0);
uprs.insertRow();
```

Sie könnten genauso mit den Spaltennummern arbeiten:

```
uprs.updateString(1, "Kona");
uprs.updateInt(2, 150);
uprs.updateFloat(3, 10.99);
uprs.updateInt(4, 0);
uprs.updateInt(5, 0);
```

1.14.6. Beispiel für das Einfügen einer Zeile in ein ResultSet

Das folgende Beispiel zeigt, wie ein Einfügen in ein ResultSet funktionieren würde. Es funktioniert aber bei Access nicht!

```
package scrollableresultsetinsert;
import java.sql.*;

public class ScrollableResultSetInsert {
    public static void main(String args[]) {
        String url = "jdbc:odbc:coffeebreak";
        Connection con;
        Statement stmt;
      try {
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      } catch(java.lang.ClassNotFoundException e) {
        System.err.print("ClassNotFoundException: ");
        System.err.println(e.getMessage());
      }
      try {
        con = DriverManager.getConnection(url, "myLogin", "myPassword");
        stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                       ResultSet.CONCUR_UPDATABLE);
        ResultSet uprs = stmt.executeQuery("SELECT * FROM COFFEES");
        uprs.moveToInsertRow();
        uprs.updateString("COF_NAME", "Kona");
        uprs.updateInt("SUP_ID", 150);
        uprs.updateFloat("PRICE", 10.99f);
        uprs.updateInt("SALES", 0);
        uprs.updateInt("TOTAL", 0);
        uprs.insertRow(); //Fehler bei Access : ArrayIndexOutOfBounds
        uprs.updateString("COF_NAME", "Kona_Decaf");
        uprs.updateInt("SUP_ID", 150);
        uprs.updateFloat("PRICE", 11.99f);
        uprs.updateInt("SALES", 0);
        uprs.updateInt("TOTAL", 0);
        uprs.insertRow();
        uprs.beforeFirst();
        System.out.println("Tabelle COFFEES nach der Ergaenzung:");
        while (uprs.next()) {
            String name = uprs.getString("COF_NAME");
            int id = uprs.getInt("SUP_ID");
            float price = uprs.getFloat("PRICE");
            int sales = uprs.getInt("SALES");
            int total = uprs.getInt("TOTAL");
            System.out.print(name + "   " + id + "   " + price);
            System.out.println("   " + sales + "   " + total);
        }

        uprs.close();
        stmt.close();
        con.close();

      } catch(SQLException ex) {
        System.err.println("SQLException: " + ex.getMessage());
      }
    }
}
```

1.14.7. Löschen einer Zeile

Das Löschen einer Zeile aus den ResultSet und der DBMS geschieht ähnlich wie das Einfügen:

```
uprs.absolute(4);
uprs.deleteRow();
```

Sinnvoll kann es auch sein, die ResultSets mit der DBMS zu synchronisieren. Dies geschieht mit folgender Anweisung:

```
Statement stmt = con.createStatement(
                          ResultSet.TYPE_SCROLL_SENSITIVE,
                          ResultSet.CONCUR_UPDATABLE);
ResultSet uprs = stmt.executeQuery(
                       "SELECT COF_NAME, PRICE FROM COFFEES");
uprs.absolute(4);
Float price1 = uprs.getFloat("PRICE");
//. . .
uprs.absolute(4);
uprs.refreshRow();
Float price2 = uprs.getFloat("PRICE");
if (price2 > price1) {
  //. . .
}
```

1.15. Batch Updates

Die Idee ist recht einfach:
da der Verbindungsaufbau und das Senden und Empfangen von Meldungen von der DBMS zeitaufwendig ist, solltenmehrere Anweisungen an die DBMS zusammengefasst werden.

Das Hinzufügen von Anweisungen zu einem Batch geschieht denkbar einfach:

```
con.setAutoCommit(false);
Statement stmt = con.createStatement();
stmt.addBatch("INSERT INTO COFFEES " +
        "VALUES('Amaretto', 49, 9.99, 0, 0)");
stmt.addBatch("INSERT INTO COFFEES " +
        "VALUES('Hazelnut', 49, 9.99, 0, 0)");
stmt.addBatch("INSERT INTO COFFEES " +
        "VALUES('Amaretto_decaf', 49, 10.99, 0, 0)");
stmt.addBatch("INSERT INTO COFFEES " +
        "VALUES('Hazelnut_decaf', 49, 10.99, 0, 0)");
int [] updateCounts = stmt.executeBatch();
```

Sinnvollerweise wird ein solcher Batch durch

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

ergänzt. Damit können Sie Batch Updates analog zu Transaktionen auffassen.

Ein Batch Update kann natürlich auch Exceptions werfen:

`SQLException` und `BatchUpdateException`

Hier ein Beispiel, wie Sie Informatione über die Ursache erfahren können:

```
      try {
        // Updates ....
        ....
      } catch(BatchUpdateException b) {
        System.err.println("SQLException: " + b.getMessage());
        System.err.println("SQLState:  " + b.getSQLState());
        System.err.println("Message:  " + b.getMessage());
        System.err.println("Vendor:  " + b.getErrorCode());
        System.err.print("Updates:  ");
        int [] updateCounts = b.getUpdateCounts();
        for (int i = 0; i < updateCounts.length; i++) {
                System.err.print(updateCounts[i] + "   ");
        }
      }
```

Auf dem Server / der CD finden Sie ein vollständiges Beispiel zu Batch Updates.
Beachten Sie, dass das Beispiel mit JDBC-ODBC für Access nicht funktioniert!

Hier das Listing:

```
package batchupdates;
import java.sql.*;
public class BatchUpdateBeispiel {
    public static void main(String args[]) {
        Connection con;
        Statement stmt;
        String url = "jdbc:odbc:coffeebreak";
        try {
          System.out.println("JDBC ODBC Driver");
          Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        } catch(java.lang.ClassNotFoundException e) {
          System.err.print("ClassNotFoundException: ");
          System.err.println(e.getMessage());
        }
        try {
            con = DriverManager.getConnection(url, "", "");
            stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
            ResultSet.CONCUR_UPDATABLE);
            con.setAutoCommit(false);
stmt.addBatch("INSERT INTO COFFEES VALUES('Amaretto', 49, 9.99, 0, 0)");
stmt.addBatch("INSERT INTO COFFEES VALUES('Hazelnut', 49, 9.99, 0, 0)");
stmt.addBatch("INSERT INTO COFFEES VALUES('Amaretto_decaf',49,10.99,0,0)");
stmt.addBatch("INSERT INTO COFFEES VALUES('Hazelnut_decaf',49,10.99,0,0)");
            int [] updateCounts = stmt.executeBatch();
            ResultSet uprs = stmt.executeQuery("SELECT * FROM COFFEES");
            System.out.println("Tabelle COFFEES nach den Mutationen:");
            while (uprs.next()) {
                String name = uprs.getString("COF_NAME");
                int id = uprs.getInt("SUP_ID");
                float price = uprs.getFloat("PRICE");
                int sales = uprs.getInt("SALES");
                int total = uprs.getInt("TOTAL");
                System.out.print(name + "\t\t " + id + "\t " + price);
                System.out.println(" " + sales + " " + total);
            }
            System.out.println("\n");
            uprs.close();
            stmt.close();
            con.close();

        } catch(BatchUpdateException b) {
            System.err.println("SQLException: " + b.getMessage());
            System.err.println("SQLState: " + b.getSQLState());
            System.err.println("Message: " + b.getMessage());
            System.err.println("Vendor: " + b.getErrorCode());
```

```
            System.err.print("Anzahl Updates: ");
            int [] updateCounts = b.getUpdateCounts();
            for (int i = 0; i < updateCounts.length; i++) {
                System.err.print(updateCounts[i] + " ");
            }
            System.err.flush();
            // falls ein Fehler in der obigen Schleife auftritt:
        } catch(SQLException ex) {
            System.err.println("SQLException: " + ex.getMessage());
            System.err.println("SQLState: " + ex.getSQLState());
            System.err.println("Message: " + ex.getMessage());
            System.err.println("Vendor: " + ex.getErrorCode());
        }
    }
}
```

1.15.1. Die SQL3 Datentypen

Die neuen Datentypen in SQL3 geben den relationalen Datenbanken mehr Flexibilität. Beispielsweise können nun die Spalten vom Datentyp BLOB (binary large object, beispielsweise Class Dateien oder EXE Dateien oder Bilder).

Zusätzlich wurden neu CLOBs definiert (character large objects), also Dateien oder Objekte, welche zeichenorientiert sind, aber viele Zeichen enthalten.

Der ARRAY Datentyp ermöglicht den Einsatz ganzer Arrays als Spalten.

Schliesslich werden in SQL3 User Defined Datatypes (UDT) unerstützt, so dass Sie eigene Datentypen definieren können.

Die Methoden mussten entsprechend angepasst werden:

SQL3 Datentyp	**getXXX Methode**	**setXXX Methode**	**updateXXX Methode**
BLOB	getBlob	setBlob	updateBlob
CLOB	getClob	setClob	updateClob
ARRAY	getArray	setArray	updateArray
Structured Type	getObject	setObject	updateObject
REF (structured type)	getRef	setRef	updateRef

Das folgende Programmfragment zeigt den Einsatz von Arrays:

```
ResultSet rs = stmt.executeQuery(
    "SELECT NOTEN FROM STUDENTEN   WHERE ID = 2238");
rs.next();
Array scores = rs.getArray("NOTEN");
```

Falls Sie eine Datenbank haben, bei der diese Datentypen unterstützt werden, können Sie auch CLOBs einsetzen:

```
Clob notes = rs.getClob("NOTES"); // Notizdatei
PreparedStatement pstmt = con.prepareStatement(
    "UPDATE MARKETS SET COMMENTS = ? WHERE SALES < 1000000",
    ResultSet.TYPE_SCROLL_INSENSITIVE,
    ResultSet.CONCUR_UPDATABLE);
pstmt.setClob(1, notes);
```

Structured Types könnten Sie definieren als:

```
CREATE TYPE PLANE_POINT
(
          X FLOAT,
          Y FLOAT
)
```

und folgendermassen einsetzen:

```
ResultSet rs = stmt.executeQuery(
    "SELECT POINTS FROM PRICES  WHERE PRICE > 3000.00");
while (rs.next()) {
  Struct point = (Struct)rs.getObject("POINTS");
  // ...
}
```

Falls Sie eigene Datentypen einsetzen könnten:

```
CREATE TYPE MONEY AS NUMERIC(10, 2)
```

Um Daten vom Typ `MONEY` zu lesen benutzen Sie `ResultSet.getBigDecimal` oder `CallableStatement.getBigDecimal`;
um ein `MONEY` Objekt, zu speichern verwenden Sie `PreparedStatement.setBigDecimal`.

1.16. Standard Extension Features

Die Neuerungen in JDBC, die im Moment in Diskussion sind, umfassen:

- **JNDI tm for Naming Databases**
  Das Java tm Naming and Directory Interface tm (JNDI) gestattet es Ihnen statt die Namen der URL direkt uns Programm einzugeben, auf einfache Art und Weise auf einen Verzeichnisdienst zuzugreifen und die Angaben dort zu holen. Sie könnten dies auch mittels Properties erreichen. Aber der Einsatz des JNDI ist universeller.

- **Connection Pooling**
  Ein Connection Pooling gestattet es Ihnen eine Verbindung mehrfach zu benutzen. Damit reduziert sich der Overhead um Einiges.

- **Distributed Transaction Support**
  Java unterstützt ein Two Phase Commit Protokoll in Java Transaction API (JTA). Dies vereinfacht den Einsatz von JDBC in Enterprise Java Beans.

1.17. Anhang 1 - SQL

Im Folgenden besprechen wir einige der grundlegenden Konzepte von SQL. Dabei geht es uns mehr um Grundlegendes, die Kenntnis einiger grundlegender Befehle.

1.17.1. Einrichten der Datenbank

Diese Übung lässt sich mit jedem Datenbank-Managementsystem realisieren, sofern SQL (Standard) unterstützt wird.

Was machen Sie im Folgenden?

1.17.1.1. Anlegen der SQLIntro Datenbank

Sie legen eine Demo Datenbank an, mit deren Hilfe wir verschiedene SQL Eigenschaften untersuchen und üben werden

Die Datenbank heisst : SQLIntro

Benutzen Sie zum Anlegen der Datenbank KEINEN Wizzard oder Assistenten!

1.17.1.2. Tabellen

In der Demo Datenbank SQLIntro müssen Sie nun verschiedene Tabellen anlegen:
Als erstes die Tabellen Mitarbeiter und Abteilung.

1.17.1.3. Anlegen der Tabellen

Die Tabelle Abteilung besteht aus den Feldern:

Abteilungs-Nummer : Feldname in der Tabelle ohne Bindestrich
Abteilungs-Name : wie oben
Abteilungs-Lokation : wie oben

Die Abteilungs-Nummer müssen Sie als Primärschlüssel definieren!

Die Tabelle Mitarbeiter besteht aus den Feldern:

Mitarbeiter-Nummer
Mitarbeiter-Name
Mitarbeiter-Funktion
Mitarbeiter-Vorgesetzter
Eintrittsdatum
Gehalt
Kommission (Verkaufspersonen erhalten eine Kommission, bei den andern Mitarbeitern bleibt das Feld leer)
Abteilungs-Nummer

1.17.1.4. Eingabe von Testdaten

Geben Sie in die zwei Tabellen je einige Testdaten ein.

Abteilungs-Tabelle:

10	Buchhaltung	New York
20	Forschung	Dallas
30	Verkauf	Chicago
40	Betrieb	Boston

Die Tabelle Mitarbeiter : (Beispiel)

7369	Smith	Gehilfe		7902	17-Dec-80	800.00	0.00	20
7902	Ford	Buchhalter	7566	03-Dec-81	1'300.00	0.00	20	
7566	Jones	Manager	7839	02-Apr-81	2'975.00	0.00	20	
7829	King	Präsident		17-Nov-81	5'000.00	0.00	10	

...

Achten Sie darauf, dass

- Die Abteilungsnummern mit denen aus der Tabelle Abteilung übereinstimmen.
- Die Vorgesetzten der Mitarbeiter jeweils auch erfasst werden

1.17.2. Select Befehl

Syntax:

SELECT einige Daten (Attribute, Spalten der Tabellen)
FROM eine oder mehrere Tabellen

SELECT steht immer am Anfang der Abfrage, FROM folgt DIREKT nach dem SELECT.

Die einfachste aller Abfragen wählt einfach alle Attribute aus:
SELECT * from Abteilung;

1.17.2.1. Anlegen einer Abfrage

Legen Sie eine Abfrage an.
Vorgehen:
Suchen Sie die Befehle zum kreieren einer Datenbank: CREATE DATABASE ...

Vervollständigen Sie das gezeigte SQL Statement:
Verwenden Sie dabei die Attributnamen IHRER Tabelle!

SELECT AbteilungsNummer, Abteilungsname, AbteilungsOrt FROM ABTEILUNG;

1.17.2.2. Führen Sie die Abfrage aus

Sie sehen in der Menüliste eine Rubrik "Abfrage". Darin finden Sie den Startbefehl für die Abfrage.

Führen Sie die obige Abfrage aus!

Welches Ergebnis erhalten Sie? Die Tabelle wird einfach Zeile für Zeile ausgegeben.
Wenn Sie wollen, können Sie die Abfrage abspeichern. Es lohnt sich aber kaum!

1.17.2.3. Kurzform dieser Abfrage

SELECT * FROM ABTEILUNG;

Testen Sie diese Abfrage! Stimmt die Behauptung?

1.17.3. Anlegen einer Tabelle

Lassen Sie das SQL Fenster offen, oder öffnen Sie es erneut mit Hilfe einer Abfrage.

Sie können Tabellen DIREKT mit SQL anlegen:

Befehls-Syntax:

```
CREATE TABLE <Tabellenname> (
        <Attribut1>    <Atrribut1 Typ>,
        <Attribut 2>   <Attribut2 Typ>,
        ...,
        <Attribut k>   <Attributk Typ>);
```

Was brauchen Sie noch?

1.17.3.1. Datentypen

Jedes Datenbanksystem unterstützt nur bestimmte Datentypen.

Finden Sie, mit Hilfe von HELP die Liste der möglichen Attribut-Typen. Geben Sie einfach SQL Datentypen als Frage ein.

1.17.3.2. Anlegen der Tabelle

Legen Sie eine Tabelle Abteilung_2 an mit genau dem selben Aufbau wie Abteilung, aber ausschliesslich mit Hilfe des CREATE TABLE Befehls!

Wie können Sie angeben, dass die Abteilungs-Nummer nicht null sein darf (als Index)?

1.17.3.3. Automatische Datenprüfung

Da Sie Attribute vom Typus Datum, Nummer und Text haben, muss das Datenbanksystem bei einer Daten-Eingabe automatisch überprüfen, ob diese Datentypen mit den vordefinierten übereinstimmen.

Testen Sie dies, indem Sie unterschiedliche Datums-Formate verwenden:
Geben Sie einmal 1-1-98, dann 1-Jan-98, ... ein.
Was passiert?

Dezimalstellen:
Wie geben Sie an, dass die Kommission auf zwei Kommastellen genau berechnet wird?

Testen Sie die Eingabe mit unterschiedlichen Daten:
Geben Sie einmal 12.12, dann 12,12, dann 12.00, 12 , ... ein.
Was passiert?

1.17.3.4. Datenkomprimierung

Intern verwenden die meisten Datenbanksysteme spezielle Algorithmen, um das Datenvolumen auf ein Minimum zu begrenzen.

1.17.4. INSERT

Daten werden in SQL wie folgt eingegeben:

Syntax:

```
INSERT INTO <Tabellennane>
VALUES (<Attribut1>,<Attribut2>, ....,<Attribut k>);
```

Beispiel:

```
INSERT INTO ABTEILUNG
VALUES (60, 'Personal', 'San Franzisco');
```

Öffnen Sie wieder ein SQL Fenster.

Ergänzen Sie eine Ihrer Tabellen mit mehreren Werten.

Kehren Sie zur Abfrage zurück oder geben Sie direkt im SQL Fenster eine Abfrage ein, die verifiziert, dass die Dateneingabe auch wirksam ist.

1.17.5. Commit

Viele Datenbanksysteme arbeiten mit einem speziellen Mechanismus, bei dem die Daten nicht direkt in die Datenbank geschrieben werden:

Zuerst stehen die Daten in einem Buffer
Erst wenn der Benutzer die Transaktion bestätigt (COMMIT;) , werden die Daten auch echt geschrieben.

Wie verfährt ORACLE?

1.17.6. Datenbank Abfragen mit Hilfe von SQL

1.17.6.1. Auswählen einzelner Attribute

1.17.6.1.1. Syntax

SELECT <Attribut>, <Attribut>, ...<Attribut>
FROM <Tabelle>;

Beispiel:
Select AbteilungsName, AbteilungsNummer
From Abteilung;

Ausgabe:

Abteilungsname	AbteilungsNummer
Buchhaltung	10
Verkauf	30
...	

1.17.6.2. Auswählen mit Einschränkungen

1.17.6.2.1. Syntax

SELECT <Attribut>, <Attribut>, ...<Attribut>
FROM<Tabelle>
WHERE <Attribut> = <Konstante>;

Beispiel:
Select AbteilungsName, AbteilungsNummer
From Abteilung
Where AbteilungsNummer = 10;

Ausgabe:

Abteilungsname	AbteilungsNummer
Buchhaltung	10

1.17.6.3. Auswählen mit mehreren Einschränkungen

1.17.6.3.1. Syntax 1

SELECT <Attribut>, <Attribut>, ...<Attribut>
FROM <Tabelle>
WHERE <Attribut1> = <Konstante1> *AND* <Attribut2> = <Konstante2>;

Beispiel:
Select AbteilungsName, AbteilungsNummer
From Abteilung
Where AbteilungsNummer = 10 AND Lokation = "Bern";

Ausgabe:
Abteilungsname AbteilungsNummer
--- ---
Buchhaltung 10

Falls die Buchhaltung tatsächlich in Bern ist dh. falls es eine Zeile in der Tabelle Abteilung gibt, die wie folgt aussieht:
...
Buchhaltung 10 Bern
...

1.17.6.3.2. Syntax 2

SELECT <Attribut>, <Attribut>, ...<Attribut>
FROM <Tabelle>
WHERE <Attribut1> = <Konstante1> *OR* <Attribut2> = <Konstante2>;

Beispiel:
Select AbteilungsName, AbteilungsNummer
From Abteilung
Where AbteilungsNummer = 20 OR Lokation = "Bern";

Ausgabe:
Abteilungsname AbteilungsNummer
--- ---
Buchhaltung 10
Public Relations 50

Falls die Public Relations tatsächlich in Bern ist dh. falls es eine Zeile in der Tabelle Abteilung gibt, die wie folgt aussieht:
...
Public Relations 50 Bern
Buchhaltung 10 New York
...

1.17.6.4. Auswählen mit negierter Einschränkungen

1.17.6.4.1. Syntax

SELECT <Attribut>, <Attribut>, ...<Attribut>
FROM <Tabelle>
WHERE <Attribut1> != <Konstante1>;

Beispiel:
Select AbteilungsName, AbteilungsNummer
From Abteilung
Where AbteilungsNummer = 10 AND Lokation != "Bern";
Ausgabe:
Abteilungsname AbteilungsNummer
--- ---
Buchhaltung 10

Hier wird das obige Resultat nur geliefert, falls die Buchhaltung NICHT in Bern ist.

1.17.6.5. Auswählen aus Bereichen

1.17.6.5.1. Syntax 1

SELECT <Attribut>, <Attribut>, ...<Attribut>
FROM <Tabelle>
WHERE <Attribut1> BETWEEN <Konstante1> AND <Konstante2>;

Beispiel:
Select AbteilungsName, AbteilungsNummer
From Abteilung
Where AbteilungsNummer between 10 AND 30;

Ausgabe:

Abteilungsname	AbteilungsNummer
Buchhaltung	10
Personal	20
Verkauf	30

1.17.6.5.2. Syntax 2

SELECT <Attribut>, <Attribut>, ...<Attribut>
FROM <Tabelle>
WHERE <Attribut1> IN (<Konstante1>,<Konstante2>,...<KonstanteX>);

Beispiel:
Select AbteilungsName, AbteilungsNummer
From Abteilung
Where AbteilungsNummer in (10, 30, 50);

Ausgabe:

Abteilungsname	AbteilungsNummer
Buchhaltung	10
Verkauf	30

falls es keine Abteilung mit der Nummer 50 gibt.

1.17.6.6. Vergleich mit einem Zeichenmuster

1.17.6.6.1. Syntax

SELECT <Attribut1>,...<AttributK>
FROM <Tabelle>
WHERE <Attribut> LIKE <Muster>;

<Muster>:
Beispiel: '__R%'

Semantik:
_ : Platzhalter
% : Folge von null oder mehr Zeichen
R : Zeichen, welches im Attribut an 3ter Stelle stehen muss

1.17.6.7. Reihenfolge der Attribute

1.17.6.7.1. Syntax

SELECT <Attribut1>,...<AttributK>
FROM <Tabelle>
...
ORDER BY <AttributX> {DESC };

Dabei muss <AttributX> in der Select Attributliste vorkommen
Falls der Zusatz DESC hinter dem Ordnungskriterium steht, dann wird absteigend sortiert, sonst aufsteigend.

Beispiel:
Select AbteilungsName, AbteilungsNummer
From Abteilung
Where AbteilungsNummer in(10,20, 30)
ORDER BY AbteilungsNummer;

Ausgabe:

Abteilungsname	AbteilungsNummer
Buchhaltung	10
Personal	20
Verkauf	30

Select AbteilungsName, AbteilungsNummer
From Abteilung
Where AbteilungsNummer in (10,20, 30)
ORDER BY AbteilungsNummer DESC;

Ausgabe:

Abteilungsname	AbteilungsNummer
Verkauf	30
Personal	20
Buchhaltung	10

1.17.6.8. Auswahl nur unterschiedlicher Zeilen

1.17.6.8.1. Syntax

SELECT {DISTINCT} <Attribut1>,...<AttributK>
FROM <Tabelle>;

Beispiel:

Abteilungsname	AbteilungsNummer	Lokation
Buchhaltung	10	Bern
Personal	20	Bern
Verkauf	30	Berlin

```
Select DISTINCT Lokation
From Abteilung
Where AbteilungsNummer  in(  10,20, 30)
ORDER BY    AbteilungsNummer;
```

Ausgabe:

```
Lokation
--------------------
Bern
Berlin
```

1.17.6.9. Aufgabe:

Wie sieht die Auswahl aus, wenn zusätzlich die Abteilungsnummer ausgewählt wird

```
Select AbteilungsName, AbteilungsNummer
From Abteilung
Where AbteilungsNummer  in(  10,20, 30)
ORDER BY    AbteilungsNummer DESC;
```

Ausgabe:

Abteilungsname	AbteilungsNummer
Verkauf	30
Personal	20
Buchhaltung	10

1.17.7. JOIN Abfragen

Wir haben zwei Tabellen:
Mitarbeiter und Abteilungen

Frage:
Wie wo arbeitet der Angestellte mit dem Namen "Peter Schmutz"?

Wie benötigen offensichtlich zwei Abfragen:
1) Abfrage der Mitarbeiter Tabelle:
in welcher Abteilung arbeitet der Mitarbeiter
Ergebnis:
Abteilungsnummer

2) Abfrage der Abteilungstabelle:
wo ist die Abteilung?
Ergebnis:
Abteilungslokation

Kombinieren wir beide Abfragen, dann erhalten wir mit Hilfe einer JOIN Abfrage das selbe Ergebnis:

```
SELECT    MitarbeiterName, Lokation
FROM      Mitarbeiter, Abteilung
WHERE     MitarbeiterName = "Peter Schmutz"
AND       Mitarbeiter.AbteilungsNummer = Abteilung.AbteilungsNummer;
```

1.17.8. Gruppenfunktionen

1.17.8.1. Syntax

SELECT Feldliste
FROM Tabelle
WHERE Kriterien
[GROUP BY Gruppenfeldliste]

Beispiel:

```
SELECT    AbteilungsNummer, MAX(Gehalt)
FROM      Mitarbeiter
GROUP BY  AbteilungsNummer;
```

Ergebnis:

```
AbteilungsNummer   MAX(Gehalt)
-----------------  -----------
10                 5'000.00
20                 3'000.00
...
```

Aufgabe:
Gruppieren Sie nach mindestens ZWEI Kriterien, wobei Sie als erstes Gruppenbruchfeld COUNT(*), als zweites AVG(Gehalt) wählen

1.17.8.2. Klausel "Having"

1.17.8.2.1. Syntax

SELECT Feldliste
FROM Tabelle
WHERE Auswahlkriterien
GROUP BY Gruppenfeldliste
[HAVING Gruppenkriterien]

Beispiel:

```
Select AbteilungsName, SUM(Gehalt), COUNT(*), AVG(Gehalt)
From   Mitarbeiter, Abteilung
Where Mitarbeiter.Abteilung = Abteilung.AbteilungsNummer
Group By      AbteilungsName, Funktion
HAVING        COUNT(*) >=2;
```

Semantik:
Ich will alle Gruppen, mit mindestens zwei Mitarbeitern und von diesen Gruppen will ich auch noch die Abteilung (als Text), die Funktion der Mitarbeiter, das Total der Gehälter und das durchschnittliche Gehalt.

1.17.9. SubQueries

Sie können fast überall wo eine <Konstante> steht auch einen SubQuery plazieren.

Beispiel:

```
Select MitarbeiterName, Mitarbeiter.Funktion
From Mitarbeiter
Where Funktion =
( Select Funktion FROM Mitarbeiter WHERE MitarbeiterName = "Müller");
```

1.17.10. Ändern gespeicherter Daten

SQL stellt drei Befehle zur Verfügung, mit deren Hilfe gespeicherte Daten verändert werden können:

- **UPDATE**
- **INSERT**
- **DELETE**

1.17.10.1. Update

Syntax

```
UPDATE Tabelle
SET        NeuerWert
WHERE      Kriterien;
```

Beispiel:

```
UPDATE Mitarbeiter
SET        Gehalt = Gehalt + 100
WHERE      Funktion = "Hilfsarbeiter";
```

Syntax Spezialfall

```
UPDATE Tabelle
SET        NeuerWert
```

Beispiel:

```
UPDATE     Mitarbeiter
SET        Gehalt = Gehalt * 1.02;
```

Jeder Mitarbeiter erhält zwei Prozent mehr Gehalt!

1.17.10.2. Insert

Syntax

Abfrage zum Anfügen mehrerer Datensätze:

```
INSERT INTO Ziel [IN ExterneDatenbank] [(Feld1[, Feld2[, ...]])]
SELECT [Quelle.]Feld1[, Feld2[, ...]
FROM Tabellenausdruck
```

Abfrage zum Anfügen eines einzelnen Datensatzes:

```
INSERT INTO Ziel [(Feld1[, Feld2[, ...]])]
VALUES (Wert1[, Wert2[, ...])
```

Beispiel:
Tabelle Mitarbeiter:
MitarbeiterNummer, MitarbeiterName,Abteilung

```
INSERT INTO Mitarbeiter (MitarbeiterNummer, MitarbeiterName,Abteilung)
SELECT MitarbeiterNummer, MitarbeiterName, Abteilung
FROM TestDaten
WHERE Funktion = "Hilfsarbeiter";
```

1.17.10.3. Delete

Syntax

```
DELETE [Tabelle.*]
FROM Tabelle
WHERE Kriterien
```

Beispiel:

```
DELETE FROM     Mitarbeiter
WHERE           AbteilungsNummer = 40;
```

Die Abteilung 40 wurde gelöscht (alle Mitarbeiter der Abteilung 40);
Erläuterungen
In der Klausel DELETE FROM ist die Tabelle namentlich angegeben, aus der Sie eine Zeile oder gleichj eine Menge von Zeilen (zum Beispiel : DELETE FROM ABTEILUNG;) streichen wollen.

Die WHERE Klausel wird nur bei Bedarf verwendet. Eine DELETE FROM Klausel löscht ALLE Zeilen einer Tabelle (da ja keine Einschränkungen gemacht werden).

Die WHERE Klauseln bei allen Befehlen in SQL verdeutlicht, dass SQL für die Datenverwaltung die gleiche Syntax verwendet wie für die Datenabfrage.

1.17.11. Dynamische Änderungen der Datenbankbeschreibung

Oracle und andere SQL basierten Datenbanken unterstützen SQL Befehle, die für eine dynamische Änderung der Datenbankstruktur benötigt werden.

- ALTER TABLE ADD - neue Spalten in eine vorhandene Tabelle einfügen
- ALTER TABLE MODIFY - eine bestehende Spalte ändern

Beispiel:
Sie haben eine SQL Datenbank und speichern die Daten in einem speziellen Format ab (zum Beispiel in Oracle : number(3) not null).

Aus der Anwendung heraus zeigt es sich, dass zum Beispiel die Jahreszahl an Stelle von zwei Stellen vielleicht doch vier Stellen haben sollte.

Sie müssen also Ihre Datenbank anpassen und hoffen dabei, keine Daten zu verlieren.

Hier der (Oracle) CREATE TABLE Befehl:

```
CREATE TABLE Projekt (   PROJNR NUMBER(3) NOT NULL,
                         PNAME CHAR(5),
                         BUDGET NUMBER(7,2) );
```

Sie haben also keine Probleme Projekte mit einem Projektbudget bis (7,2) zu erfassen:

```
INSERT INTO Projekt (101, 'eCommerce', 96000);
INSERT INTO Projekt (102, 'WWW', 82000);
INSERT INTO Projekt (103, 'MediaDB',15000);
```

Im Laufe des Projektes MediaDB erkennen Sie, dass es praktisch wäre, den Projektleiter mit zu erfassen.

Sie möchten also ein Feld PROJMGR in die Tabelle einfügen.

Lösung:
```
ALTER TABLE Projekt ADD (ProjMgr number(3));
```

Damit verknüpfen Sie die Tabelle Projekt mit der Tabelle Mitarbeiter, in der jeder Mitarbeiter eine eindeutige, maximal dreistellige Mitarbeiternummer hat.

```
UPDATE Projekt
SET     ProjMgr = 901
WHERE PNAME='eCommerce';
```

Die weiteren Projekte ordnen wir analog zu.

Nach der ersten Abklärungsphase im Projekt MediaDB stellen wir fest:
Das Budget müsste korrigiert werden auf ungefähr das 10 - fache:

Unser Feld in der Tabelle sieht aber nicht genug Stellen vor dem Komma vor!

Falls wir versuchen :

```
UPDATE Projekt
SET BUDGET =105000
WHERE PROJNO = 103;
```

Dann erhalten wir die Fehlermeldung:

ERROR : value larger than specified precision allows for this column;

Wir müssen also unsere Tabelle ÄNDERN

```
ALTER TABLE Projekt MODIFY (BUDGET NUMBER(8,2));
```

1.17.12. Alternative Benutzersichten

Oft haben wir das Problem, dass nach einer Normalisierung die erklärenden Texte zu einer Nummer fehlen. Wir müssen dann mit Hilfe von komplexen Joins die Daten wieder zusammen suchen.

Um das zu vereinfachen wurde ein neues Konstrukt : die VIEW definiert.

Ziele und Zweck von VIEWS:

- Vereinfachen von Zugriffen auf die Daten
- Sorgen von Datenunabhängigkeit
- errichten eines Datenschutzes

Ein einfaches Beispiel:

```
CREATE VIEW Mitarbeiter102 AS
        SELECT MitarbeiterNummer, MitarbeiterName, MitarbeiterFunktion
        FROM Mitarbeiter
        WHERE Abteilung =102;
```

Jetzt können wir mit Hilfe dieser Sicht Daten abfragen:

```
Select * from Mitarbeiter102;
```

1.17.13. Datensicherheit und Gemeinsame Datennutzung

Grössere, mehrbenutzerfähige Datenbanken benötigen Hilfsmittel zur Verwaltung der Zugriffsrechte auf unterschiedlichen Ebenen:

1. Auf der Ebene der Datenbank als Ganzes:
   - Welche Benutzer dürfen überhaupt auf die Datenbank zugreifen (lesen, schreiben, mutieren)
2. Auf der Stufe der Tabellen:
   - Welche Benutzer dürfen auf welche Tabellen zugreifen (lesen, schreiben, mutieren)
3. Welcher Benutzer darf welchem Benutzer welche Rechte weiter geben?

ORACLE und andere grosse Datenbanksysteme verwenden dafür folgende Befehle:

- GRANT - Erteilen von Zugriffsberechtigungen auf Tabellen und Views an andere Benutzer
- REVOKE - Zurücknahme der vergebenen Zugriffsrechte

SIE sind der Eigentümer aller von Ihnen angelegten Tabellen. Nur Sie als Eigentümer dürfen diese Tabellen benutzen. Sie haben aber das Recht (und in der Regel die Pflicht) andern Benutzern Zugriffsrechte zu erteilen.

1.17.13.1. GRANT ON TO

GRANT dient der Erteilung von Zugriffsrechten. Allerdings muss man unterscheiden:

- Sie können JEMANDEM Zugriffsrechte erteilen :
  GRANT .... TO
- Sie können jemandem Zugriffsrechte AUF Tabellen, oder Sichten erteilen :
  GRANT ... ON

1.17.13.1.1. Beispiel

GRANT SELECT
ON MITARBEITER
TO Schild;

Sie haben die Tabelle Mitarbeiter (in einer Datenbank Test.dbs) angelegt und erlauben dem Benutzer Schild den lesenden Zugriff auf die Mitarbeiter-Tabelle.
Zugriffsberechtigungen können in unterschiedlichen Kombinationen erteilt (und später wieder weg genommen werden):

Zugriffsberechtigung für Tabellen	für Views
SELECT	SELECT
INSERT	INSERT *)
UPDATE	UPDATE *)
DELETE	DELETE *)
ALTER	
INDEX	
CLUSTER	

*) : in vielen Fällen ist es Unsinn mit Hilfe von Views Tabellen zu verändern.
Grund:
Wenn die Sicht nur einzelne Datenfelder / Attribute berücksichtigt, dann führt ein UPDATE, INSERT oder DELETE sofort zu inkonsistenten Daten. Wie soll gelöscht werden, wenn sich die Sicht nur auf ein Attribut einer Tabelle bezieht?

1.17.13.2. Zusammenfassung

Die Zugriffsberechtigungen werden gemäss dem Schema

```
GRANT    <Operation>
ON       <Tabelle oder View>
TO       <Benutzer>
```

vergeben.

1.17.13.3. Bemerkung : Zugriffsrechte in Access

Auszug aus der Hilfe Funktion:

"ANSI SQL-Funktionen, die von Microsoft Jet-SQL nicht unterstützt werden

Microsoft Jet-SQL unterstützt folgende Funktionen von ANSI SQL nicht:

· Anweisungen für Sicherheitsfunktionen wie COMMIT, **GRANT** und LOCK.
· Verweise auf die Aggregatfunktion DISTINCT. Microsoft Jet-SQL läßt z.B. den Ausdruck SUM(DISTINCT Spaltenname) nicht zu.
· Den Abschnitt LIMIT TO nn ROWS, der verwendet wird, um die Anzahl der Zeilen einzuschränken, die von einer Abfrage zurückgegeben werden. Nur der WHERE-Abschnitt kann verwendet werden, um den Umfang einer Abfrage einzuschränken."

1.17.13.4. Konsequenzen

Seien Sie vorsichtig beim Einsatz von Access in einer Multiuser-Umgebung oder bei Applikationen, in denen LOCK, COMMIT oder GRANT vorkommen könnten!

1.17.13.5. REVOKE ON FROM

Die Umkehrung von GRANT ist REVOKE. Der Aufbau ist gleich wie bei GRANT.

1.17.13.5.1. Beispiel

```
REVOKE   INSERT
ON       MITARBEITER
FROM     Schild;
```

Jetzt darf "Schild" keine Daten mehr in die Tabelle MITARBEITER (oder die Datensicht MITARBEITER) einfügen.

1.17.14. Performance - Indizierung und Clusterung

Relationale Datenbanken sind recht komplexe Gebilde und müssen in der Regel GB von Daten verwalten.

Es lohnt sich also über Performance nach zu denken (oder das Geld für VIEL Hardware aus zu geben).

Welche Möglichkeiten bieten sich an?

4. Schaffung eines Index (invertierte Liste) für den schnellen Zugriff auf bestimmte Zeilen in einer Tabelle
5. Aufbau eines Clusters aus mehreren Tabellen für optimale Verknüpfungsvorgänge

1.17.14.1. Indizierung

Werden Indices für Ihre Daten geschaffen, dann hilft dies der Datenbank beim Auffinden bestimmter Zeilen auf den Seiten in der Datenbank (physische Datenbereiche werden in Pages / Seiten aufgeteilt). Sie kennen das vom Telefonbuch her: ohne Register und alphabetische Anordnung hätten wir Probleme den Heinrich Meier in Luzern im Telefonbuch zu finden.

1.17.14.2. Beispiel

```
CREATE INDEX
MITARBEITER_IDX
ON  MITARBEITER(PERSONALNUMMER);
```

CREATE INDEX legt einen neuen Index für **eine bereits vorhandene** Tabelle an.

1.17.14.3. Anmerkung zu Access

Das Microsoft Jet-Datenbankmodul unterstützt die Verwendung von CREATE INDEX oder einer anderen der DDL-Anweisungen (DDL = Data Definition Language) nicht für Datenbanken, die nicht auf dem Microsoft Jet-Datenbankmodul basieren (außer zur Erstellung eines Pseudoindexes auf einer mit ODBC verknüpften Tabelle). Verwenden Sie statt dessen die Create-Methoden für Datenzugriffsobjekte. Weitere Informationen finden Sie unter "Bemerkungen".

1.17.14.3.1. Syntax

```
CREATE [ UNIQUE ] INDEX Index
ON Tabelle (Feld [ASC|DESC][, Feld [ASC|DESC], ...])
[WITH { PRIMARY | DISALLOW NULL | IGNORE NULL }]
```

Die CREATE INDEX-Anweisung besteht aus folgenden Teilen:

Teil	Beschreibung
Index	Der Name des Indexes, der angelegt werden soll.
Tabelle	Der Name der existierenden Tabelle, die den Index enthalten wird.
Feld	Der Name eines oder mehrerer zu indizierender Felder. Sie erstellen einen Einzelfeldindex, indem Sie den Feldnamen in Klammern hinter dem jeweiligen Tabellennamen angeben. Sie erstellen einen Mehrfelderindex, indem Sie den Namen aller Felder angeben, die in die Indexdefinition aufgenommen werden

sollen. Sie können Indizes für eine absteigende Sortierung erstellen, indem Sie das reservierte Word DESC (für descending = absteigend) verwenden. Andernfalls legen Indizes eine aufsteigende Sortierreihenfolge fest.

Bemerkungen

Mit dem reservierten Wort UNIQUE können Sie verhindern, daß zu einem Index gehörende Felder in unterschiedlichen Datensätzen gleiche Werte enthalten.
Im optionalen WITH-Abschnitt können Sie Regeln zur Überprüfung von Daten vorschreiben:

- Mit der Option DISALLOW NULL können Sie verhindern, daß neu angelegte Datensätze in den Feldern, die zum Index gehören, Null-Einträge enthalten.
- Mit der Option IGNORE NULL können Sie dafür sorgen, daß Datensätze, die in den indizierten Feldern Null-Werte enthalten, in den Index aufgenommen werden.
- Mit dem reservierten Wort PRIMARY können Sie die Felder, die den Index definieren sollen, als Primärschlüssel festlegen. Da dies impliziert, daß der Schlüssel eindeutig ist, können Sie auf die Angabe des reservierten Worts UNIQUE verzichten.

Sie können mit CREATE INDEX auch einen Pseudoindex für eine verknüpfte Tabelle aus einer ODBC-Datenquelle (z.B. von SQL Server) erstellen, die noch keinen Index hat. Sie benötigen keine Berechtigungen oder keinen Zugriff auf den Remote Server, um einen Pseudoindex zu erstellen. Der Remote-Datenbank ist der Pseudoindex nicht bekannt, und sie wird dadurch auch nicht verändert. Sowohl für verknüpfte als auch für systemeigene Tabellen verwenden Sie dieselbe Syntax. Dies ist besonders hilfreich, wenn Sie einen Index für eine Tabelle erstellen, die normalerweise aufgrund eines fehlenden Indexes schreibgeschützt wäre.

Sie können mit der ALTER TABLE-Anweisung einer Tabelle auch einen Einzelfeldindex oder einen Mehrfelderindex hinzufügen, und mit der ALTER TABLE-Anweisung oder der DROP-Anweisung können Sie einen Index entfernen, der mit ALTER TABLE oder CREATE INDEX erstellt wurde.

Anmerkung Verwenden Sie das reservierte Wort PRIMARY nicht, wenn Sie einen neuen Index für eine Tabelle erstellen, die bereits einen Primärschlüssel hat, da dies einen Fehler verursachen würde.

1.17.14.4. Löschen eines Index

1.17.14.5. Beispiel

DROP INDEX MITARBEITER_IDX;
Generell gilt:

Mit DROP wird eine Tabelle, Index, Benutzersicht, ... aus der Datenbank gelöscht.

1.17.14.6. Anmerkung zu Access

Das Microsoft Jet-Datenbankmodul unterstützt die Verwendung von DROP oder einer anderen der DDL
-Anweisungen (DDL = Data Definition Language) nicht für Datenbanken, die nicht auf dem Microsoft Jet-Datenbankmodul basieren. Verwenden Sie statt dessen die Delete-Methoden für Datenzugriffsobjekte.

1.17.14.6.1. Syntax

```
DROP {TABLE Tabelle | INDEX Index ON Tabelle}
```

Die DROP-Anweisung besteht aus folgenden Teilen:

Teil	Beschreibung
Tabelle	Der Name der Tabelle, die gelöscht werden soll, oder der Tabelle, aus der ein Index gelöscht werden soll.
Index	Der Name des Indexes, der aus Tabelle gelöscht werden soll.

Bemerkungen
Sie müssen die Tabelle schließen, bevor Sie sie löschen oder einen Index daraus entfernen können.
Sie können einen Index auch mit der ALTER TABLE-Anweisung aus einer Tabelle löschen.
Sie können mit CREATE TABLE eine Tabelle und mit CREATE INDEX oder ALTER TABLE einen Index erstellen. Zum Ändern einer Tabelle verwenden Sie ALTER TABLE.

1.17.15. Clusterbildung

Clusterbildung ist eine weitere Methode zur Erhöhung der Systemleitung.
Wenn Sie Cluster aufbauen, so weiss das Datenbanksystem, dass Zeilen aus verschiedenen Tabellen physisch dicht beieinander (in einem physischen Cluster von benachbarten Speicherseiten) abgespeichert werden sollten.Dadurch werden JOIN Verknüpfungen, die fast in jeder Datenbankabfrage auftreten, beschleunigt.

1.17.15.1. Anmerkung zu Access

Access und viele andere Datenbanken unterstützen Cluster NICHT!

1.17.15.2. Beispiel

```
CREATE CLUSTER ABTEILUNG_MITARBEITER(ABTEILUNGS_NUMMER);

ALTER CLUSTER ABTEILUNG_MITARBEITER
ADD TABLE ABTEILUNG
WHERE ABTEILUNG.ABTEILUNGS_NUMMER =
ABTEILUNG_MITARBEITER.ABTEILUNGS_NUMMER;

ALTER CLUSTER ABTEILUNG_MITARBEITER
ADD TABLE MITARBEITER
WHERE MITARBEITER.ABTEILUNGS_NUMMER =
ABTEILUNG_MITARBEITER.ABTEILUNGS_NUMMER;
```

Wir definieren also erst den Cluster und fügen anschliessend die Tabellen hinzu. Im Cluster haben wir das JOIN Datenfeld als Attribut angeben.