In diesem Kursteil
- Kursübersicht
- Modul 1 : Java Database Connectivity JDBC
  - Modul Einleitung
  - JDBC Driver
  - java.sql Package
  - JDBC Abläufe
  - Verbindungsaufbau mittels JDBC
  - JDBC Driver
  - JDBC Connection & Statements
  - Mapping : SQL Datentypen auf Java
  - Praktische Übung
  - Quiz
  - Zusammenfassung

Java in Verteilte Systeme - JDBC

1.1. Kursübersicht

Der Kurs ***Java in Verteilten Systemen*** ist eine Einführung in die Technoligien
- JDBC : Datenbankzugriff
- RMI : verteilte Objektsysteme
- Objektserialisierung

mit dem Ziel, Ihnen Wissen zu vermitteln, welche Sie befähigen wird, verteilte Anwendungen zu entwickeln. Dieser Kurs beschreibt Technologien, mit deren Hilfe Sie verteilte Anwendungen entwickeln können, basierend auf Java™ Application Programming Interfaces (API).

Dieses Skript beschränkt sich auf den JDBC Teil. Sie finden das ganze Skript unter den Einführungskurs-Unterlagen

Voraussetzung für diesen Kurs sind Kenntnisse im Bereich *Java Programmierung*, *Objekt-Orientiertes Design und Analyse* und mindestens teilweise folgende praktischen Erfahrungen:
- Entwicklung von Java Applikationen
- Grundkenntnisse in Datenbanken
- Grundkenntnisse von SQL

Sie sollten idealerweise auch bereits Erfahrungen in der objektorientierten Programmierung haben.

1.1.1. Lernziele

Nach dem Durcharbeiten dieser Kursunterlagen sollten Sie in der Lage sein
- unterschiedliche Technologien für die Programmierung verteilter Systeme in Java zu kennen und zu vergleichen
- einfache Datenbank-Anwendungen zu schreiben
- Remote Methode Invocation Applikationen zu schreiben und Daten mittels Objektserialisierung langfristig zu speichern.
- einfache Java IDL Applikatione zu schreiben

© J.M.Joller

1.2. Modul 1 : Java Database Connectivity JDBC

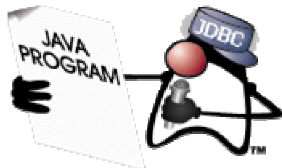
In diesem Modul

- Modul 1 : Java Database Connectivity (JDBC)
- Modul Einleitung
- JDBC Driver
- Das java.sql Package
- JDBC Abläufe
- DB Verbindungsaufbau mittels JDBC
- JDBC Treiber
- JDBC Connection
- JDBC Anweisungen
- Abbildung von SQL Datentypen auf Java Datentypen
- Einsatz des JDBC APIs
- Datenbank Design
- Applets
- Praktische Übung
- Quiz
- Zusammenfassung

1.2.1. Einleitung

Das JDBC API ist ein Set von Interfaces, mit deren Hilfe in Datenbankanwendungen die Details des Datenbankzugriffs isoliert werden können. Die JDBC Interfaces gestatten es dem Entwickler, sich auf die eigentlich Applikation zu konzentrieren, also sicherzustellen, dass die Datenbankabfragen korrekt formuliert sind und die Datenbank korrekt designed ist.

Mit JDBC kann der Entwickler gegen ein Interface entwickeln, mit den Methoden und Datenfeldern, die das Interface zur Verfügung stellt. Der Entwickler braucht sich dabei nicht gross darum zu kümmern, dass es Interfaces vor sich hat. Die Trieber Anbieter stellen Klassen zur Verfügung, welche die Interfaces implementieren. Daher wird der Programmierer eigentlich gegen einen Treiber programmieren.



Das JDBC API gestattet es den Entwicklern, beliebige Zeichenketten direkt an den Treiber zu übergeben, also low level Programmierung. Daher kann der Entwickler auch SQL Spracheigenschaften einer bestimmten Implementation verwenden, nicht nur SQL Konstrukte des ANSI SQL Standards.

© J.M.Joller

1.2.1.1. Lernziele
Nach dem Durcharbeiten dieses Moduls sollten Sie in der Lage sein:
- zu erklären, was JDBC ist.
- die fünf wichtigsten Aufgaben bei der Programmierung eines JDBC Programms aufzählen können.
- zu erklären, wie der JDBC Driver mit dem JDBC Driver Manager zusammenhängt.
- zu erklären, wie die Datenbank-Datentypen in Java Datentypen umgewandelt werden.
- unterschiedliche Architekturen für verteilte Systeme, beispielsweise die zwei-Tier und drei-Tier Architektur gegeneinander abzugrenzen, speziell im Zusammenhang mit JDBC.
- einfache JDBC Datenbank Applikationen zu schreiben und JDBC Probleme zu lösen.

1.2.1.2. Referenzen
Teile dieses Moduls stammen aus der JDBC Spezifikation
- " The JDBC ™ 1.2 Specification" bei Sun http://java.sun.com/products/jdbc/
- SQL können Sie beispielsweise in ***The Practical SQL Handbook*** von Emerson, Darnovsky und Bowman (Addison-Wesley, 1989) nachlesen.

1.2.2. JDBC Driver
Ein JDBC Driver ist eine Sammlung von Klassen, welche die JDBC Interfaces implementieren, also die Interfaces, welche es Java Programmen erlauben, auf Datenbanken zuzugreifen. Jeder Datenbank Treiber muss mindestens eine Klasse zur Verfügungstellen, welche das `java.sql.Driver` Interface implementiert. Diese Klasse wird von der generischen `java.sql.DriverManager` Klasse benutzt, falls diese einen Treiber benötigt, um auf eine bestimmte Datenbank mittels eines Uniform Resource Locators (URL) zuzugreifen. JDBC sieht ähnlich aus wie ODBC. Daher kann man JDBC auch zusammen mit ODBC effizient implementieren. Der Overhead ist gering, die Implementation also effizient.

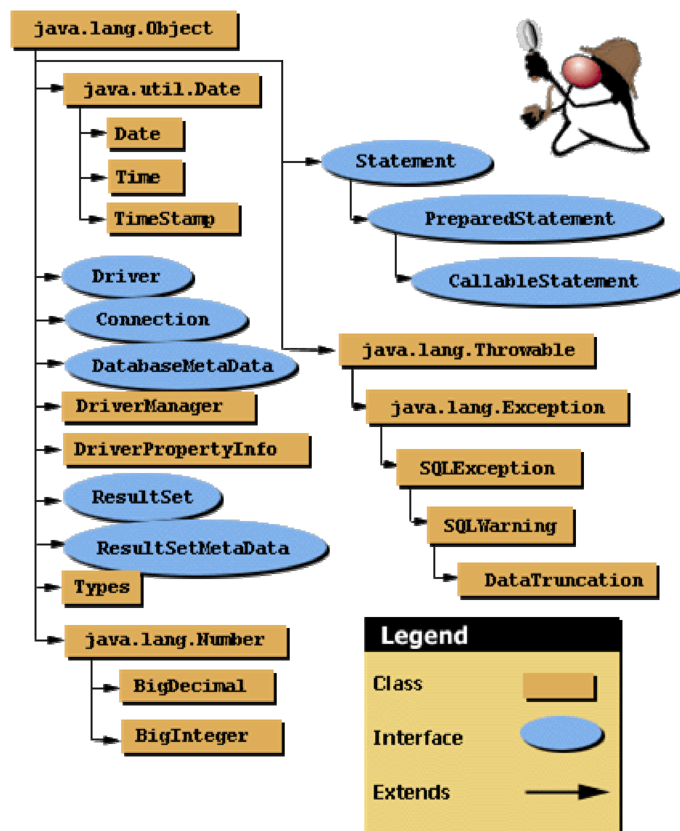
Eine einzige Java Applikation (oder ein Applet) kann gleichzeitig auf mehrere Datenbanken mittels eines oder mehreren Treibern zugreifen.

© J.M.Joller

1.2.3. Das *java.sql* Package

Zur Zeit gibt es acht Interfaces im Zusammenhang mit JDBC:

1. Driver
2. Connection
3. Statement
4. PreparedStatement
5. CallableStatement
6. ResultSet
7. ResultSetMetaData
8. DatabaseMetaData



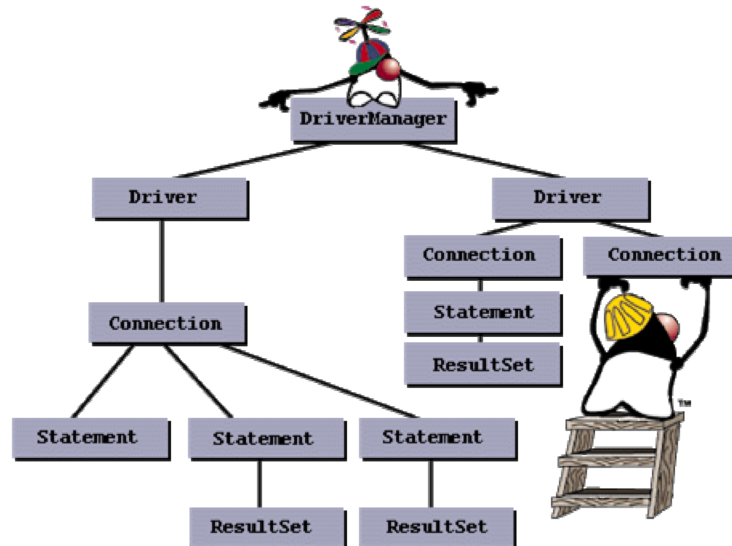
Die obigen Interfaces müssen alle implementiert werden. Aber es liegt an Ihnen zu entscheiden, welche der definierten Methoden Sie für Ihre Applikation konkret benötigen.

© J.M.Joller

1.2.4. JDBC Abläufe

Jedes der Interfaces stellt Ihnen bestimmte Verbindungsoptionen für eine Verbindung zu einer Datenbank, zum Ausführen von SQL Anweisungen und zum Bearbeiten der Ergebnisse zur Verfügung.

Eine URL Zeichenkette wird an die getConnection() Methode der DriverManagers übergeben. Der DriverManager sucht einen Driver und damit kann eine Verbindung eine Connection hergestellt werden.



Mit der Verbindung können Sie eine Datenbank Anweisung, ein Statement, absetzen. Falls das Statement ausgeführt wird, mit der executeQuery() Methode, liefert die Datenbank einen Ergebnisset, den ResultSet, da beispielsweise eine Datenbankabfrage in der Regel mehr als einen Datensatz zurückliefert. Im anderen Extremfall ist der ResultSet leer, weil kein Datensatz die Abfragekriterien erfüllt, eine Mutation durchgeführt wurde oder ein neuer Datensatz in die Datenbank eingefügt wurde.

© J.M.Joller

1.2.5. Verbindungsaufbau mittels JDBC Interface

Nun betrachten wir einige typischen Aufgaben, die Sie mit JDBC erledigen, falls Sie als Programmierer damit arbeiten. Wir werden mit MS-Access, Textdateien, Excel und ähnlichen ODBC Datenbanken arbeiten. Aber eigentlich spielt dies keine zentrale Rolle, da wir mit ODBC arbeiten können und damit die Datenbank selber keine Rolle spielt. Die im Text verwendeten Driver und SQL DB ist : My-SQL mit derm JDBC Driver von imaginary.

```
package connecttodbms;

import java.sql.*;

public class ConnectToCoffeeBreakDB {

      public static void main(String args[]) {
            String url = "jdbc:odbc:coffeebreak";
            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, "", "");
                  System.out.println("con.close() -
                                          Verbindung schliessen");
                  con.close();
            } catch(SQLException ex) {
                  System.err.println("SQLException: Verbindungsaufbau " +
                                     ex.getMessage());
            }
      }
}
```

Im folgenden Beispiel sehen Sie, wie Sie beispielsweise in einer JDBC Applikation eine Driver Instanz bilden können, anschliessend ein Verbindungsobjekt kreieren, ein Statementobjekt bilden und eine Abfrage ausführen und anschliessend die Ergebnisse der Abfrage im ResultSet Objekt bearbeiten:

```
import java.sql.*;
import com.imaginary.sql.msql.MsqlDriver;// miniSQL Treiber,
// falls Sie nicht mit ODBC arbeiten, wie oben

public class JDBCBeispiel {

   public static void main (String args[]) {

      if (args.length < 1) {
      System.out.println ("Usage:");
      System.out.println ("java JDBCBeispiel <db server hostname>");
      System.exit (1);
    }

    try {
      // Instanz des iMsqlDrivers bilden (mySQL z.B.)
      new com.imaginary.sql.msql.MsqlDriver ();
```


© J.M.Joller

```
      // Definition der "url"
      String url = "jdbc:msql://" + args[0] + ":1112/FlugreservationsDB";

      // Verbindungsaufbau mit Hilfe des DriverManager
      Connection conn = DriverManager.getConnection (url);

      // kreieren eines Statement Objekts
      Statement stmt = conn.createStatement ();

      // Query ausführen (mit dem Statement Objekt)
      // und Definition des ResultSet Objekts
      ResultSet rs = stmt.executeQuery ("SELECT * from FlugzeugTyp");
      // Ausgabe der Ergebnisse - Zeile für Zeile
      while (rs.next()) {
        System.out.println ("");
        System.out.println ("Flugzeugtyp:           "
        + rs.getString (1));
        System.out.println ("First Class Plätze:    "
        + rs.getInt (2));
        System.out.println ("Business Class Plätze: "
        + rs.getInt (3));
        System.out.println ("Economy Class Plätze:    "
        + rs.getInt (4));
      }

  } catch (SQLException e) {
      e.printStackTrace();
  }
 }
}
```

Falls Sie das Programm starten, werden die Inhalte der Tabelle FlugzeugTyp angezeigt:

```
Flugzeugtyp:           B747
First Class Plätze:    30
Business Class Plätze: 56
Economy Class Plätze:  350

Flugzeugtyp:           B727
First Class Plätze:    24
Business Class Plätze: 0
Economy Class Plätze:  226

Flugzeugtyp:           B757
First Class Plätze:    12
Business Class Plätze: 0
Economy Class Plätze:  112

Flugzeugtyp:           MD-DC10
First Class Plätze:    34
Business Class Plätze: 28
Economy Class Plätze:  304
```

wobei ich vorgängig die Access Datenbank und die Tabelle angelegt und Daten eingegeben habe.

© J.M.Joller

1.2.6. Kreieren eines JDBC Driver Objekts

Das Treiberobjekt können Sie entweder direkt im Programm oder aber als Programm Property beim Starten angeben:

```
java –D jdbc.drivers= com.imaginary.sql.msql.MsqlDriver Abfrage
```

Wie dies genau geschieht werden wir noch anschauen. Der obige JDBC Treiber stammt von der Firma Imaginary und ist für die mSQL (mini-SQL) Datenbank entwickelt worden.
Um mit einer Datenbank kommunizieren zu können, muss auch jeden Fall eine Instanz des JDBC Treibers vorhanden sein. Der Treiber agiert im Hintergrund. Er behandelt alle Anfragen an die Datenbank.

Sie brauchen keine Refernz des Treibers treibers, es genügt also, wenn Sie einfach ein Objekt kreieren:

```
new com.imaginary.sql.msql.MsqlDriver();
```

Der Treiber existiert nach dieser Anweisung, sofern er erfolgreich geladen werden kann. Daher können wir auch einfach mit

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

die Klasse direkt laden. Im ersten Fall wird der Konstruktor ein statisches Objekt kreieren, welches im System erhalten bleibt. Eine Referenz selbst wird nicht benötigt. Alle Methoden der Klasse DriverManager sind `static`, können also direkt mit dem Klassenpräfix aufgerufen werden. Der Driver ist auch für das Registrieren des Drivers zuständig. Dies können Sie auch explizit durchführen:

```
Driver drv = DriverManager.getDriver(url);
DriverManager.registerDriver(drv);
```

Wenn Sie dies nochmals explizit machen, wird der selbe Driver zweimal registriert, wie Sie mit `DriverManager.getDrivers()` abfragen können:

```
// fakultatives Registrieren (wird implizit gemacht)
Driver drv = DriverManager.getDriver(url);
DriverManager.registerDriver(drv);
for (Enumeration drvs=DriverManager.getDrivers();drvs.hasMoreElements(); )
      System.out.println(drvs.nextElement().getClass());
```

Dies liefert folgende Ausgabe:

```
class sun.jdbc.odbc.JdbcOdbcDriver
class sun.jdbc.odbc.JdbcOdbcDriver
```

Wie Sie sehen, wurde der selbe Driver zweimal geladen. Gleichzeitig sehen Sie auch, dass mehr als ein Driver geladen werden kann, sogar mehrere Driver, die den Zugriff auf die selbe Datenbank gestatten, beispielsweise einmal über JDBC, einmal über ODBC.

Als Property, auf der Kommandozeile, sieht die Spezifikation mehrerer Driver so aus:

```
jdbc.drivers = com.imaginary.sql.msql.MsqlDriver:Acme.wonder.driver
```


© J.M.Joller

Properties setzt man mit der -D Option des Java Interpreters oder mittels der -J Option beim Appletviewer.

Beispiel:
```
java -D jdbc.drivers= com.imaginary.sql.msql.MsqlDriver:Acme.wonder.driver
```

Falls Ihr Programm versucht, eine Verbindung mit einer Datenbank über JDBC aufzunehmen, wird der erste Driver der gefunden wird eingesetzt. Die Reihenfolge ist folgendermassen festgelegt:

1. die Driver der Property Angabe werden von links nach rechts geprüft und der erste mögliche davon eingesetzt.
2. dann werden die bereits geladenen Driver geprüft, in der Reihenfolge, in der sie geladen wurden.

Falls ein Driver mit untrusted Code geladen wurde, also nicht sicher sein muss, wird der Driver nicht weiterverwendet, ausser das Programm hat den Driver neu von der selben Quelle geladen.

Wie auch immer, sobald der Driver geladen ist, ist er dafür verantwortlich, sich selbst beim Driver Manager zu registrieren. Schauen wir uns den Mini-SQL Driver von Imaginary an:

```
/* Copyright (c) 1997 George Reese */
package com.imaginary.sql.msql;
import java.sql.Connection;
import java.sql.Driver;
import java.sql.DriverManager;
import java.sql.DriverPropertyInfo;
import java.sql.SQLException;
import java.util.Properties;
/**
 * The MsqlDriver class implements the JDBC Driver interface from the
 * JDBC specification.  A Driver is specifically concerned with making
 * database connections via new JDBC Connection instances by responding
 * to URL requests.<BR>
 * Last modified 97/10/28
 * @version @(#) MsqlDriver.java 1.4@(#)
 * @author George Reese (borg@imaginary.com)
 */
public class MsqlDriver implements Driver {
    /****************** Static methods and attributes *****************/
    // The static constructor does according to the JDBC specification.
    // Specifically, it creates a new Driver instance and registers it.
    static {
      try {
          new MsqlDriver();
      }
      catch( SQLException e ) {
          e.printStackTrace();
      }
    }
    /****************** Instance methods and attributes *****************/
    /**
     * Constructs an MsqlDriver instance.  The JDBC specification requires
     * the driver then to register itself with the DriverManager.
     * @exception java.sql.SQLException an error occurred in registering
     */
    public MsqlDriver() throws SQLException {
      super();
```


© J.M.Joller

```
  DriverManager.registerDriver(this);
}
/**
 * Gives the major version for this driver as required by the JDBC
 * specification.
 * @see java.sql.Driver#getMajorVersion
 * @return the major version
 */
public int getMajorVersion() {
  return 1;
}
/**
 * Gives the minor version for this driver as required by the JDBC
 * specification.
 * @see java.sql.Driver#getMinorVersion
 * @return the minor version
 */
public int getMinorVersion() {
  return 0;
}

/**
 * The getPropertyInfo method is intended to allow a generic GUI tool
 * to discover what properties it should prompt a human for in order to
 * get enough information to connect to a database.Note that depending
 * on the values the human has supplied so far, additional values
 * may become necessary, so it may be necessary to iterate though
 * several calls to getPropertyInfo.
 * @param url The URL of the database to connect to.
 * @param info A proposed list of tag/value pairs that will be sent on
 *          connect open.
 * @return An array of DriverPropertyInfo objects describing possible
 *        properties.This array may be an empty array if no properties
 *          are required.
 * @exception java.sql.SQLException never actually thrown
 */
public DriverPropertyInfo[] getPropertyInfo(String url,Properties info)
throws SQLException {
  return new DriverPropertyInfo[0];
}

/**
 * Returns true if the driver thinks that it can open a connection
 * to the given URL.  In this case, true is returned if and only if
 * the subprotocol is 'msql'.
 * @param url The URL of the database.
 * @return True if this driver can connect to the given URL.
 * @exception java.sql.SQLException never actually is thrown
 */
public boolean acceptsURL(String url) throws SQLException {
  if( url.length() < 10 ) {
      return false;
  }
  else {
      return url.substring(5,9).equals("msql");
  }
}

/**
 * Takes a look at the given URL to see if it is meant for this
 * driver.  If not, simply return null.  If it is, then go ahead and
 * connect to the database.  For the mSQL implementation of JDBC, it
```


© J.M.Joller

```
 * looks for URL's in the form of <P>
 * <PRE>
 *     jdbc:msql://[host_addr]:[port]/[db_name]
 * </PRE>
 * @see java.sql.Driver#connect
 * @param url the URL for the database in question
 * @param p the properties object
 * @return null if the URL should be ignored, a new Connection
 * implementation if the URL is a valid mSQL URL
 * @exception java.sql.SQLException an error occurred during connection
 * such as a network error or bad URL
 */
public Connection connect(String url,Properties p)throws SQLException {
  String host, database, orig = url;
  int i, port;

  if( url.startsWith("jdbc:") ) {
      if( url.length() < 6 ) {
        return null;
      }
      url = url.substring(5);
  }
  if( !url.startsWith("msql://") ) {
      return null;
  }
  if( url.length() < 8 ) {
      return null;
  }
  url = url.substring(7);
  i = url.indexOf(':');
  if( i == -1 ) {
      port = 1114;
      i = url.indexOf('/');
      if( i == -1 ) {
        throw new SQLException("Invalid mSQL URL: " + orig);
      }
      if( url.length() < i+1 ) {
        throw new SQLException("Invalid mSQL URL: " + orig);
      }
      host = url.substring(0, i);
      database = url.substring(i+1);
  }
  else {
      host = url.substring(0, i);
      if( url.length() < i+1 ) {
        throw new SQLException("Invalid mSQL URL: " + orig);
      }
      url = url.substring(i+1);
      i = url.indexOf('/');
      if( i == -1 ) {
        throw new SQLException("Invalid mSQL URL: " + orig);
      }
      if( url.length() < i+1 ) {
        throw new SQLException("Invalid mSQL URL: " + orig);
      }
      try {
        port = Integer.parseInt(url.substring(0, i));
      }
      catch( NumberFormatException e ) {
        throw new SQLException("Invalid port number: " +
                          url.substring(0, i));
      }
```


© J.M.Joller

```
            database = url.substring(i+1);
        }
        return new MsqlConnection(orig, host, port, database, p);
    }

    /**
     * Returns information noting the fact that the mSQL database is not
     * SQL-92 and thus cannot support a JDBC compliant implementation.
     */
    public boolean jdbcCompliant() {
        return false;
    }
}
```

Der JDBC Treiber der Firma Imaginary kreiert eine Instanz von sich selbst, oben im statischen Block und registriert sich entweder automatisch oder explizit.

Nach dem JDBC Treiber müssen wir nun die Datenbank angeben, auf die wir zugreifen möchten. Dies geschieht mit Hilfe einer URL, welche den Datenbanktyp angibt. Dabei handelt es sich um eine URL ähnliche Notation, also keine echte URL.

```
jdbc:subprotocol:parameters
```

Beispiel:

```
jdbc:odbc:FlugreservationsDB
```

dabei steht `subprotocol` einen bestimmten Datenbank Verbindungsmechanismus, im Beispiel also ODBC. Die Parameter hängen vom DBMS und dem Protokoll ab:

```
// konstruiere die URL für den JDBC Zugriff
String url = new String ("jdbc:msql://" + args[0]+":1112/TicketingDB");
```

In diesem Fall haben wir auch noch den Port und die Datenbank angegeben, da wir nicht über ODBC die entsprechenden Informationen beschaffen können. In diesem Beispiel ist das Subprotokoll

```
msql
```

Falls wir an Stelle des Mini-SQL Treibers über ODBC auf die Datenbank zugreifen möchten, hätten wir

```
jdbc:odbc:Object.TicketingDB
```

angeben können. Aber auch ein Netzwerk-Protokoll könnte verwendet werden:

```
jdbc:nisnaming:Ticketing-Info
```

Der JDBC URL Mechanismus liefert ein Framework, so dass unterschiedliche Driver eingesetzt werden können. Jeder Treiber muss einfach die URL Syntax verstehen, da er irgend etwas daraus machen muss!

© J.M.Joller

1.2.7. Die JDBC Verbindung - Connection

Nachdem wir eine "URL" zur Datenbank besitzen und damit den Datenbanktypus festgelegt haben, müssen wir die Verbindung zur Datenbank aufbauen. Dies geschieht mit einem

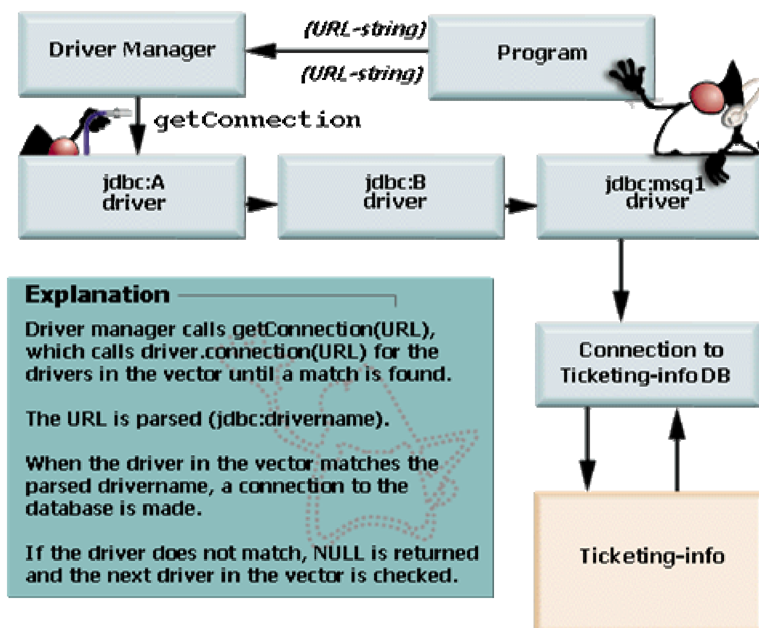
```
java.sql.Connection
```

Objekt, indem die

```
java.sql.DriverManager.getConnection()
```

Methode des JDBC Driver Manager aufgerufen wird.

Schematisch, gemäss original Sun Dokumentation:




```
// Verbindungsaufbau zur Datenbank mittels des
// DriverManager
Connection conn = DriverManager.getConnection (url);
```

Der Ablauf ist :

1. Der `DriverManager` ruft die Methode `Driver.getConnection()` auf, für jeden registrierten Driver, mit der URL, also einer Netzwerkadresse, als Parameter.

2. Falls der Driver eine Verbindung herstellen kann, dann liefert er ein Connection Objekt zurück, sonst null.

Ein `DriverManager` löst die URL Referenz in der `getConnection()` Methode auf. Falls der `DriverManager null` zurückliefert, versucht er gleich mit dem nächsten Treiber eine Verbindung aufzubauen, bis schliesslich die Liste erschöpft ist, oder eine Verbindung hergestellt werden konnte. Die Verbindung selbst besteht nicht zwischen dem Treiber und der Datenbank - die Verbindung ist die Aufgabe der Implementation des `Connection` Interfaces.

© J.M.Joller

1.2.8. JDBC Anweisungen

Um eine Abfrage an eine Datenbank zu senden, muss man zuerst ein `Statement` Objekt von der `Connection.createStatement()` Methode erhalten.

```
// kreieren eines Statement Objekts
      try {
            Statement stmt = conn.createStatement ();
      catch(SQLException se) {
            System.out.println(e.getMessage() );
      }
```

Bemerkung SQLException

SQL Ausnahmen treten bei Datenbankfehlern auf. Dies kann beispielsweise eine unterbrochene Verbindung oder ein heruntergefahreren Datenbankserver sein. SQL Exceptions liefern zusätzliche Debugging Informationen:

- eine Zeichenkettenbeschreibung des Fehlers
- eine SQL Zustandsbeschreibung gemäss dem Xopen Standard
- ein anbieterspezifischer Fehlercode

1.2.8.1. Direkte Ausführung - `Statement`

Mit der Methode `Statement.executeQuery(...)` kann man die SQL Aweisung an die Datenbank senden. JDBC verändert die SQL Anweisung nicht, sondern reicht sie einfach weiter. JDBC interpretiert also SQL nicht.

```
// kreieren eines Statement Objekts
Statement stmt = conn.createStatement();
// Query ausführen (mit dem Statement Objekt)
// und Definition des ResultSet Objekts
ResultSet rs = stmt.executeQuery ("SELECT * from FlugzeugTyp");
```

1.2.8.2. Vorbereitete Ausführung - `PrepareStatement`

Die `Statement.executeQuery()` Methode liefert ein ResultSet, welches anschliessend bearbeitet werden muss. Falls die selbe Anweisung öfters verwendet werden soll, ist es vorteilhaft zuerst die `PrepareStatement` Anweisung auszuführen.

```
public static void main(String args[]) {
    System.out.println("[PrepareStatement]Start");
    String url = "jdbc:odbc:coffeebreak";
    Connection con;
    String query = "SELECT * FROM SUPPLIERS";
    Statement stmt;
    System.out.println("[PrepareStatement]Treiber laden");
    try {
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    } catch(java.lang.ClassNotFoundException e) {
      System.err.print("ClassNotFoundException: ");
      System.err.println(e.getMessage());
    }
    try {
      System.out.println("[PrepareStatement]getConnection()");
      con = DriverManager.getConnection(url, "myLogin", "myPassword");
      System.out.println("[PrepareStatement]executeQuery");
```


© J.M.Joller

```
      PreparedStatement prepStmnt = con.prepareStatement(query);
      ResultSet rs = prepStmnt.executeQuery();
      ResultSetMetaData rsmd = rs.getMetaData();
      int numberOfColumns = rsmd.getColumnCount();
      int rowCount = 1;
      while (rs.next()) {
        System.out.println("Row " + rowCount + ":  ");
        for (int i = 1; i <= numberOfColumns; i++) {
          System.out.print("   Column " + i + ":  ");
          System.out.println(rs.getString(i));
        }
        System.out.println("");
        rowCount++;
      }
      con.close();
    } catch(SQLException ex) {
      System.err.println("SQLException: Procedures werden nicht unterstützt
");
      System.err.println(ex.getMessage());
    }
  }
```

mit der Ausgabe

```
[PrepareStatement]Start
[PrepareStatement]Treiber laden
[PrepareStatement]getConnection()
[PrepareStatement]executeQuery
Row 1:
   Column 1:  49
   Column 2:  Superior Coffee
   Column 3:  1 Party Place
   Column 4:  Mendocino
   Column 5:  CA
   Column 6:  95460

Row 2:
   Column 1:  101
   Column 2:  Acme, Inc.
   Column 3:  99 Market Street
   Column 4:  Groundsville
   Column 5:  CA
   Column 6:  95199

Row 3:
   Column 1:  150
   Column 2:  The High Ground
   Column 3:  100 Coffee Lane
   Column 4:  Meadows
   Column 5:  CA
   Column 6:  93966
```

Sie müssen klar auseinanderhalten

1) auf der einen Seite "prepared":
   in diesem Fall bereiten Sie die Ausführung einer SQL Anweisung vor und führen Sie nachher mehrere Male (oder auch nur einmal) aus.
2) auf der andern Seite "stored"
   in diesemFall bereiten Sie die Ausführung auch vor. Aber die Anweisung selber wird in der Datenbank abgespeichert, 'stored'.

Zusätzlich können Sie auch SQL Anweisungen parametrisiert vorbereiten:

© J.M.Joller

```
public boolean prepStatement(Reservation obj){
      PreparedStatement prepStmnt = msqlConn.prepareStatement( "UPDATE
                        Fluege SET anzahlVerfuegbareFCSitze = ? WHERE
                        flugNummer = ?" );
prepStmnt.setInt(1,(Integer.parseInt(obj.anzahlVerfuegbareFCSitze)-1));
      prepStmnt.setLong(2, obj.FlugNr);
      int rowsUpdated = prepStmnt.executeUpdate();
      if (rowsUpdated > 0){
            return true;
      } else {
            return false;
      }
}
```

Platzhalter, Parameter des `PreparedStatement` werden einfach durchnummeriert.
Die Parameter könnten `in`, `out` oder `inout` Parameter sein. JDBC unterstützt nur `in` und `out`. `inout` werden im Rahmen von CORBA IDL berücksichtigt. `in` Variablen werden 'by value' übergeben, also wertmässig. `out` Parameter werden 'by reference', an eine Referemz übergeben.

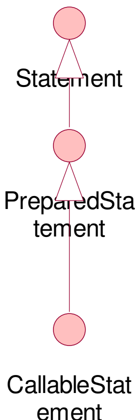
Die übergebenen Datentypen müssen mit den von SQL unterstützten Datentypen übereinstimmen:

Methode	SQL Typ(en)
setASCIIStream	verwendet einen ASCII Stream um ein LONGVARCHAR zu produzieren
setBigDecimal	NUMERIC
setBinaryStream	LONGVARBINARY
setBoolean	BIT
setByte	TINYINT
setBytes	VARBINARY oder LONGVARBINARY (abhängig von der Grösse und der möglichen Grösse von VARBINARY)
setDate	DATE
setDouble	DOUBLE
setFloat	FLOAT
setInt	INTEGER
setLong	BIGINT
setNull	NULL
setObject	das Java Objekt wird in einen SQL Datentyp konvertiert bevor es an die DB gesandt wird
setShort	SMALLINT
setString	VARCHAR oder LONGVARCHAR (abhängig von der Grösse, die der Driver VARCHAR zuordnet)
setTime	TIME
setTimestamp	TIMESTAMP
setUnicodeStream	UNICODE

1.2.8.3. Gespeicherten Anweisung - `CallableStatement`

Das dritte Interface, welches eine Erweiterung des PreparedStatement darstellt, geht davon aus, dass eine SQL Anweisung vorbereitet und in der Datenbankabgespeichert werden kann. Solche Anweisungen nennt man deswegen auch 'Stored Procedures', eben weil sie in der Datenbank abgespeichert werden.

© J.M.Joller



In JDBC definiert man `CallableStamenet` sogar etwas allgemeiner: ein `CallableStatement` gestattet die Ausführung von nicht-SQL Anweisungen gegenüber einer Datenbank. Typischerweise handelt es sich dabei um Anweisungen, welche nur von einzelnen Datenbanken unterstützt werden.

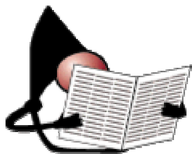
Da das Interface `CallableStatement` eine Erweiterung des `PreparedStatement` ist, können auch `in` Parameter gesetzt werden. Da `PreparedStatement` das Interface Statement erweitert, können auch dessen Methoden problemlos eingesetzt werden, beispielsweise `Statement.getMoreResults()`.

Hier ein Beispiel für eine Abfrage einer Datenbank mit einer precompiled SQL Anweisung, die Abfrage der Anzahl Plätze eines Fluges:

```
1  String planeID = "727";
2  CallableStatement querySeats = msqlConn.prepareCall("{call
      return_seats[?, ?, ?, ?]}");
   //              1, 2, 3, 4
3  try {
4      querySeats.setString(1, planeID);
5      querySeats.registerOutParameter(2, java.sql.Type.INTEGER);
6      querySeats.registerOutParameter(3, java.sql.Type.INTEGER);
7      querySeats.registerOutParameter(4, java.sql.Type.INTEGER);
8      querySeats.execute();
9      int FCSeats = querySeats.getInt(2);
10     int BCSeats = querySeats.getInt(3);
11     int CCSeats = querySeats.getInt(4);
12 } catch (SQLException SQLEx){
13          System.out.println("Abfrage schlug fehl!");
14          SQLEx.printStackTrace();
15 }
```

Bevor man eine Stored Procedure aufrufen kann, muss man explizit den Ausgabeparameter registrieren:

```
registerOutParameter(parameter, java.sqlType.<SQLDatentyp>)
```



Das Ergebnis eines solchesn Aufrufes kann eine Tabelle sein, welche mittels eines `java.sql.ResultSet` Objekts abgefragt werden kann. Die Tabelle besteht aus Zeilen und Spalten. Die Zeilen werden je nach Abfrage sortiert oder einfach sequentiell in das Ergebnisset geschrieben. Ein `ResultSet` enthält einen Zeiger auf die aktuelle Datenzeile und zeigt am Anfang vor die erste Zeile. Mit dem ersten Aufruf der next() Methode zeigt der Zeiger auf die erste Zeile, der zweite Aufruf verschiebt den Zeiger auf die zweite Zeile usw.

Das ResultSet Objekt stellt einige `set...( )` Methoden zur Verfügung, mit deren Hilfe auf die einzelnen Datenelemente in der aktuellen Zeile zugegriffen werden kann, verändernd. Auf die Spaltenwerte kann man entweder mittels Spaltennamen oder mittels Index zugreifen. In der Regel ist es besser einen Index zu verwenden. Die Indices starten, wie Sie oben bereits gesehen haben, mit 1 und werden einfach durchnummeriert. Falls man Spaltennamen

© J.M.Joller

verwendet könnte es im Extremfall mehr als eine Spalte mit demselben Namen haben. Das würde also sofort zu einem Konflikt führen.

Mit Hilfe verschiedener `get...( )` Methoden hat man Zugriff auf die Daten, zum Lesen. Die Spalten können in beliebiger Reihenfolge gelesen werden, innerhalb einer definierten Zeile.

Damit man Daten aus einem `ResultSet` Objekt herauslesen kann, müssen Sie die unterschiedlichen unterstützten Datentypen und den Aufbau des `ResultSet` kennen.

```
while (rs.next()) {
      System.out.println ("");
      System.out.println ("Flugzeugtyp:" + rs.getString (1));
      System.out.println ("Erste Klasse:" + rs.getInt (2));
      System.out.println ("Business Klasse:" + rs.getInt (3));
      System.out.println ("Economy Klasse:" + rs.getInt (4));
}
```

Bemerkung

Nachdem Sie das ResultSet Objekt gelesen haben, sind die Ergebnisse gelöscht. Sie können den `ResultSet` nur einmal lesen. Das Konzept wurde und wird aber im Moment erweitert, beispielsweise durch scrollable ResultSets. Aber eine Methode `ResultSet.numberOfRows()` gibt es auch weiterhin nicht.

Methode	Java Typ
getASCIIStream	java.io.InputStream
getBigDecimal	java.math.BigDecimal
getBinaryStream	java.io.InputStream
getBoolean	Boolean
getByte	Byte
getBytes	byte[]
getDate	java.sql.Date
getDouble	Double
getFloat	Float
getInt	Int
getLong	Long
getObject	Object
getShort	Short
getString	java.lang.String
getTime	java.sql.Time
getTimestamp	java.sql.Timestamp
getUnicodeStream	java.io.InputStream oder Unicode Zeichen

© J.M.Joller

1.2.9. Abbildung von SQL Datentypen auf Java Datentypen

In der folgenden Tabelle sind die entsprechenden Datentypen zu den SQL Datentypen ersichtlich. Die SQL Datentypen müssen allgemeiner als die Java Datentypen definiert sein, da diese für allgemeine Datenbanken gültig sein müssen.

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

© J.M.Joller

1.2.10. Generelles zum Einsatz des JDBC APIs

Die Grundidee des JDBC war es, ein Set von Interfaces zu schaffen, welche möglichst universell, also unabhängig von der speziellen Datenbank einsetzbar sind. Zusätzlich sollte das JDBC auch nicht zu sehr an ein bestimmtes DB Design oder eine Design Methodik angelehnt sein. JDBC soll also unterschiedliche Designalternativen gestatten.

Allerdings hängt das Datenbankdesign zum Teil vom JDBC Treiber ab:
- bei einem **two-tier** (zweistufigen) Design verwenden Sie Java, um direkt auf die Datenbankmethoden oder Bibliotheken zuzugreifen; Sie verwenden also datenbankspezifischen Code und Protokolle.
- bei einem **three-tier** (dreistufigen) Design verwenden Sie Java und JDBC. Die JDBC Aufrufe werden anschliessend in DBMS Methodenaufrufe umgesetzt. Der JDBC ist somit DBMS unabhängig.
- beide Architekturen können zusätzlich verallgemeinert werden, indem Microsoft's Open Database Connectivity (ODBC) eingesetzt wird. Die JDBC Aufrufe werden in ODBC Aufrufe umgesetzt und ein in der Regel plattformunabhängiger ODBC Treiber sorgt für die Datenbankmethodenaufrufe.

Zurzeit gibt es über 100 Treiber für JDBC. Sie finden die aktuelle Liste unter

http://industry.java.sun.com/products/jdbc/drivers

In der Regel werden die Treiber auch von unabhängigen Firmen angeboten. Damit hat man die Gewähr, dass diese möglichst universell einsetzbar sind. Der JDBC zu ODBC Treiber wird auch direkt mit Java mitgeliefert. Natürlich gibt es viele Firmen, die behaupten bessere Implementationen zu besitzen, beispielsweise schnellere.

© J.M.Joller

1.2.11. Datenbank Designs

Ein zweistufiges Design ist eines, bei dem der Benutzer mit einer Frontend Applikation (Client) direkt mit einer Datenbank (Server) verbunden ist.

Eine Datenbankanwendung mit einer *zweistufigen* Architektur kann JDBC auf zwei Arten einsetzen:

- ein **API Treiber** benutzt eine Bibliothek, in der Regel in C oder C++, welche für eine bestimmte Hardware- Plattform und ein bestimmtes Betriebsystem entwickelt wurde, jene für die eine Anwendung entwickelt wird. Der Client kann in irgend einer Programmiersprache geschrieben sein.
- ein **Protokoll Treiber** benutzt ein spezifisches und in der Regel Datenbank- abhängige Protokolle, welche vom Datenbankhersteller zur Verfügung gestellt werden. Der Datenbankanbieter kann beispielsweise einen TCP / IP (Transport protocol / Internet Protocol) Listener zur Verfügung stellen, der die Anfragen mehrerer Clients verwalten kann. Der Client muss dabei oberhalb TCP/IP ein Datenbank- spezifisches Protokoll einsetzen, um Abfragen an die Datenbank zu senden und die Ergebnisse zu empfangen und auszuwerten.

Ein *dreistufiges* Datenbankdesign umfasst zusätzlich einen Prozess zwischen der Endbenutzer- Applikation und dem Datenbank- Server. Das sieht auf Anhieb nach Overhead aus, also Leistungsverlust. Aber ein solches Design besitzt auch klare Vorteile:

- die Datenvalidierung wird vom mitteleren Prozess, vom Middle-Tier, übernommen.
- der Client kann mit einer einzigen Socketverbindung auf unterschiedliche Datenbanken zugreifen. Der Client kann dabei auch ein Applet in einem Browser sein.
- das Client- zum - Middle Protokoll ist unabhängig von der Datenbank. Die eigentliche Anbindung an die Datenbankist Aufgabe des middle tiers.
- im middle tier können Sie zusätzliche Funktionalitäten anbieten, beispielsweise Record Locking oder Änderungsmeldungen für einzelne Datensätze.

Datenbankanwendungen mit einem dreistufigen Design können einen Treiber einsetzen, der ausschliesslich in Java geschrieben ist. Der JDBC Treiber übersetzt die Datenbank- Methodenaufrufe in das Protokoll der mittleren Ebene. Die mittlere Ebene ist für die Auswahl der ausgewählten Datenbank zuständig.

© J.M.Joller

1.2.12. Applets

Applets waren einmal der Startpunkt für die Verbreitung von Java im Internet, speziell im Web Bereich. JDBC kann auch in Applets integriert werden. Das Hauptproblem ist in diesem Falle die Security Policy.

Applets kann man aber auch im Intranet einsetzen, um beispielsweise Firmendaten abzufragen. Aber die Applets unterscheiden sich in einigen wesentlichen Aspekten von Standardapplikationen:

Wegen der Sicherheitseinschränkungen kann ein Applet nicht auf lokale Daten eines fremden Servers zugreifen. Auch ein ODBC Zugriff über lokale Registry (ODBC) Einträge ist nicht möglich.

Auch der Overhead wegen komplexen Netzwerkverbindungen, eventuell rund um die Welt, kann zu Problemen führen.

Einige der Sicherheitsprobleme kann man mit Hilfe digitaler Signaturen und kryptografischen Methoden umgehen bzw. lösen. Aber auch die Verzeichnisdienste in heterogenen Systemen können zu grossen Problemen führen.

Falls man eine dreistufige Architektur einsetzt, ist die mittlere Ebene vorzugsweise objektorientiert, selbst wenn die Aufrufe mittels RPC in C/C++ erfolgen.

© J.M.Joller

1.2.13. Praktische Übung - Java Datenbank Connectivity

1.2.13.1. Lernziele

In dieser Übung sollten Sie lernen, wie aus einem Java Programm mittels JDBC auf eine Datenbank zugegriffen und diese abgefragt werden kann. Als Beispiel verwenden wir die Flugreservation, die wir als Interface definieren und in späteren Beispielen weiterverwenden.

Dieses Beispiel zeigt Ihnen, wie man
- auf eine SQL Datenbank zugreifen kann.
- wie man diese Datenbank abfragen kann und alle Abflughäfen bestimmen kann. Jeder Abflughafen wird dabei genau einmal ausgegeben.
- wie man die Datenbank abfragen kann, um alle Zielflughäfen zu bestimmen. Auch hier soll jeder Zielflughafen jeweils nur einmal ausgegeben werden.

- Falls der Abflughafen, der Zielflughafen und das Flugdatum bekannt ist, werden alle Flüge zwischen den zwei Orten an jenem Datum ausgegeben.

© J.M.Joller

1.2.13.2. Das Flug-Interface

Schauen wir uns als erstes das Flug Interface an. Dieses Interface spezifiziert die Methoden, welche benötigt werden, um mit einer SQL Datenbank zu kommunizieren und die oben erwähnten Daten zu lesen:

- alle Abflughäfen, jeweils nur einmal (keine Mehrfachnennungen)
- alle Bestimmungs-Flughäfen (keine Mehrfachnennungen)
- bei gegebenem Abflughafen, Destination und Abflugdatun: alle Flüge.

```
package flugreservation;

/**
 * Title:
 * Description: Das Interface beschreibt die benötigten Methoden für unser
 *              Flugreservationsbeispiel
*/

public interface Fluege {

   /**
    * Die Methode bestimmeAlleAbflughaefen liefert alle Abflughäfen als Array
    */
   String [] bestimmeAlleAbflughaefen();

   /**
    * Die Methode bestimmeAlleZielflughaefen() liefert alle Zielflughäfen als Array
    */
   String [] bestimmeAlleZielflughaefen();

  /**
   * Die Methode bestimmeAlleFluege liefert alle Flüge
   * am Datum date
   * vom Flughafen origin
   * zum Flughafen dest
   */
   FlightInfo[] bestimmeAlleFluege(String origin, String dest, String date);

  /**
   * Die Methode produceFlightString generiert eine eindeutige Zeichenkette
   * für alle Flüge
   */
   String produziereFlugString (FlugInfo flugObjekt);
}
```


© J.M.Joller

1.2.13.3. Die Verbindung zur Datenbank

Nun schauen wir uns an, wie der Verbindungsaufbau zur Datenbank aussieht.

Unser Szenario sieht folgendermassen aus: wir wollen
- den Kunden identifizieren.
- das Kundenprofil abspeichern und Informationen über den Kunden abfragen können.
- alle Flüge von...nach an einem bestimmten Tag bestimmen.
- die Sitze des Flugzeugs verwalten können.
- einen bestimmten Flug reservieren können.
- alle Informationen über einen bestimmten Flug abfragen können.

Schauen Sie sich folgendes Programmfragment an.
Frage: welche der folgenden Antworten ist korrekt?[1]
a) der Driver wird kreiert
b) die URL wird gebildet
c) eine Verbindung wird hergestellt
d) b) und c)

```
   ...
public FluegeImpl(String ServerName) {
    String url = " jdbc:odbc://"+ServerName + ":1112/FlugreservationsDB";

    try {
      mSQLcon = DriverManager.getConnection(url);
    } catch(java.sql.SQLException SQLEx) {
      System.out.println(SQLEx.getMessage() );
      SQLEx.printStackTrace();
      System.out.println("Die Datenbank antwortet nicht");
    }
  }

   ...
```

Frage: wie könnte eine SQL Abfrage aussehen, welche die Zielflughäfen bestimmt?

Lösungsskizze:

```
/**
   * Die Methode bestimmeAlleZielflughaefen() liefert alle Zielflughäfen
als Array
   */
  public String [] bestimmeAlleZielflughaefen(){
    String [] DestStaedte = null;
    Vector destStaedteV = new Vector(10, 10);
    Statement Destinationen = null;
    ResultSet rs = null;
    int j = 0;

    /* festlegen eines SQL Statements
       Abfragen mit DISTINCT Keyword garantiert, dass jedes Record,
       jeder Datensatz
       nur einmal angezeigt wird.
    */

    try {
      Destinationen = mSQLcon.createStatement();
      rs = Destinationen.executeQuery
            ("SELECT DISTINCT destStadt FROM Fluege");
    } catch (java.sql.SQLException SQLEx){
```

[1] d) ist korrekt

© J.M.Joller

```
      System.out.println(SQLEx.getMessage());
    }

    try {
      DestStaedte = new String[rs.getMetaData().getColumnCount()];
      while (rs.next()){
        destStaedteV.addElement(rs.getString(1));
      }
      rs.close();       // Result Set schliessen
    } catch (java.sql.SQLException SQLEx){
      System.out.println("Spalte gibts nicht in der Datenbank-Tabelle");
    }

    DestStaedte = new String[destStaedteV.size()];
    Enumeration OE = destStaedteV.elements();
    while(OE.hasMoreElements()){
      DestStaedte[j] = (String)OE.nextElement();
      j++;
    }
    System.out.println(DestStaedte[0]+"   "+DestStaedte[1]);
    return DestStaedte;
  }
```

Frage: gibt es wesentliche Unterschiede zwischen den zwei Methoden (bestimmen aller Abflughäfen, bestimmen aller Zielflughäfen)?

```
/**
   * Die Methode bestimmeAlleAbflughaefen liefert alle Abflughäfen als
Array
   */
  public String [] bestimmeAlleAbflughaefen(){
    // Statement Objekt ist der Gateway zur Datenbank
    Statement Origins = null;
    Vector OrigStaedteV = new Vector(10, 10);
    String [] OrigStaedte = null;
    ResultSet rs = null;
    int j = 0;

    /* mit DISTINCT werden alle Datensätze nur einmal angezeigt
     */
    try {
      Origins = mSQLcon.createStatement();
      System.out.println (Origins);
      Rs = Origins.executeQuery("SELECT DISTINCT originCity FROM Fluege");
    } catch (java.sql.SQLException SQLEx){
      System.out.println(SQLEx.getMessage());
    }

    try {
      System.out.println(rs);
      while (rs.next()){
        OrigStaedteV.addElement(rs.getString(1));
      }
      rs.close();       // Resultset
    } catch (java.sql.SQLException SQLEx){
      System.out.println("Diese Spalte gibtes nicht");
    }

    OrigStaedte = new String[OrigStaedteV.size()];
    Enumeration OE = OrigStaedteV.elements();
    while(OE.hasMoreElements()){
      OrigStaedte[j] = (String)OE.nextElement();
      j++;
    }
    return OrigStaedte;
  }

  /**
```


© J.M.Joller

```
    * Die Methode bestimmeAlleZielflughaefen() liefert alle Zielflughäfen
als Array
    */
  public String [] bestimmeAlleZielflughaefen(){
    String [] DestStaedte = null;
    Vector destStaedteV = new Vector(10, 10);
    Statement Destinationen = null;
    ResultSet rs = null;
    int j = 0;

    /* festlegen eines SQL Statements
       Abfragen mit DISTINCT Keyword garantiert, dass jedes Record / jeder
Datensatz
       nur einmal angezeigt wird.
    */

    try {
      Destinationen = mSQLcon.createStatement();
      rs = Destinationen.executeQuery("SELECT DISTINCT destStadt FROM
Fluege");
    } catch (java.sql.SQLException SQLEx){
      System.out.println(SQLEx.getMessage());
    }

    try {
      DestStaedte = new String[rs.getMetaData().getColumnCount()];
      while (rs.next()){
        destStaedteV.addElement(rs.getString(1));
      }
      rs.close();       // Result Set schliessen
    } catch (java.sql.SQLException SQLEx){
      System.out.println("Spalte mit diesem Namen gibts nicht in der
Datenbank-Tabelle");
    }

    DestStaedte = new String[destStaedteV.size()];
    Enumeration OE = destStaedteV.elements();
    while(OE.hasMoreElements()){
      DestStaedte[j] = (String)OE.nextElement();
      j++;
    }
    System.out.println(DestStaedte[0]+"   "+DestStaedte[1]);
    return DestStaedte;
  }
```

Lösung:
beide Methoden verhalten sich ähnlich!

Damit haben Sie, bis auf den FlugString, alle Methoden beieinander, um eine einfache Flugreservation zu implementieren.

Schauen Sie sich den Programmcode genau an.

Sie finden auf dem Server / der CD zusätzlich ein GUI als Prototyp, mit dem Sie eine Applikation vervollständigen können.

© J.M.Joller

Ein Problem ist im obigen Prototypen vorhanden: Sie sollten sich entweder auf deutsche oder englische Datenfeldnamen festlegen. Oben finden Sie ein schlechtes Beispiel: die Namen sind gemischt und führen bei der Implementation und der Zusammenführung des GUI Prototypen und der Methoden zu Problemen.

© J.M.Joller

1.2.14. Quiz

Zur Abrundung des Stoffes hier noch einige Fragen, die Sie ohne grössere Probleme beantworten können, falls Sie den Stoff zu diesem Thema eingehend durchgearbeitet haben.

Frage: welche der folgenden Aussagen trifft zu?

a) JDBC gestattet es einem Entwickler eine Applikation mittels Interfaces im JDBC API zu entwickeln, unabhängig davon, wie der Treiber implementiert wird. [2]
b) JDBC gestattet es einem Entwickler auf entfernte Daten genauso zuzugreifen, wie wenn alle Daten lokal wären.[3]
c) JDBC besteht aus einem Interface Set, welches es gestattet, die Applikation von der speziellen Datenbank zu trennen.[4]
d) a) und c) sind korrekt.[5]

Frage: welche der acht Interfaces aus JDBC (`Driver, Connection, Statement, PreparedStatement, CallableStatement, ResultSet, ResultSetMetatData und DatabaseMetaData`) müssen implementiert werden?

a) `Driver, Connection, Statement` und `ResultSet`[6]
b) `Driver, Connection, Statement, ResultSet, PrepareStatement, CallableStatement`, `ResultSetMetaData` und `DatabaseMetaData` [7]
c) `PreparedStatement, CallableStatement` und `ResultSetMetaData` [8]

Frage: nehmen Sie Stellung zu folgender Aussage!
Falls Sie mit einer Datenbank mittels JDBC Verbindung aufnehmen wollen, verwendet JDBC den ersten verfügbaren Treiber, um mit der URL eine Verbindung herzustellen.

Antwort:

a) trifft zu[9]
b) trifft nicht zu

---

[2] teilweise richtig
[3] nein: dies ist der Fall bei RMI, der Remote Methode Invocation
[4] diese Antwort ist teilweise richtig.
[5] genau
[6] denken Sie nochmals nach
[7] korrekt: alles muss implementiert werden, falls Sie diese benutzen wollen.
[8] nein
[9] korrekt: zuerst werden die Treiber aus der properties Liste verwendet, dann jene die bereits geladen sind.

© J.M.Joller

Frage: welches der folgenden Objekte wird eingesetzt, falls Sie mehrfach dieselbe Abfrage machen wollen?[10]

```
a) PreparedStatement
b) Statement
c) Statement.executeQuery()
```

Frage: welche der folgenden Beschreibung gilt für ein dreistufiges, *three-tiers* Modell?

a) Sie verwenden Java, um direkt auf die Datenbankmethoden oder Bibliotheken zuzugreifen; Sie verwenden also datenbankspezifischen Code und Protokolle.
b) Sie verwenden Java und JDBC. Die JDBC Aufrufe werden anschliessend in DBMS Methodenaufrufe umgesetzt. Der JDBC ist somit DBMS unabhängig.

1.2.15. Zusammenfassung

Nach dem Durcharbeiten dieses Moduls sollten Sie in der Lage sein:

- zu erklären, was man unter JDBC versteht
- welches die fünf Aufgaben sind, mit der sich der JDBC Programmierer herumschlagen muss.
- zu erklären, wie der Treiber mit dem Treiber-Manager zusammenhängt.
- wie Datenbanktypen auf Java Datentypen abgebildet werden.
- welches die Unterschiede zwischen two-tiers und three-tiers Architekturen sind.

Sie sollten in der Lage sein, JDBC in der Praxis einzusetzen und damit DBMS Abfragen oder Mutationen durchzuführen.

[10] a)

© J.M.Joller

JAVA IN VERTEILTE SYSTEME - JDBC **1**
1.1. KURSÜBERSICHT 1
1.1.1. Lernziele *1*
1.2. MODUL 1 : JAVA DATABASE CONNECTIVITY JDBC 2
1.2.1. Einleitung *2*
1.2.1.1. Lernziele 3
1.2.1.2. Referenzen 3
1.2.2. JDBC Driver *3*
*1.2.3. Das **java.sql** Package* *4*
1.2.4. JDBC Abläufe *5*
1.2.5. Verbindungsaufbau mittels JDBC Interface *6*
1.2.6. Kreieren eines JDBC Driver Objekts *8*
1.2.7. Die JDBC Verbindung - Connection *13*
1.2.8. JDBC Anweisungen *14*
1.2.8.1. Direkte Ausführung - `Statement` 14
1.2.8.2. Vorbereitete Ausführung - `PrepareStatement` 14
1.2.8.3. Gespeicherten Anweisung - `CallableStatement` 16
1.2.9. Abbildung von SQL Datentypen auf Java Datentypen *19*
1.2.10. Generelles zum Einsatz des JDBC APIs *20*
1.2.11. Datenbank Designs *21*
1.2.12. Applets *22*
1.2.13. Praktische Übung - Java Datenbank Connectivity *23*
1.2.13.1. Lernziele 23
1.2.13.2. Das Flug-Interface 24
1.2.13.3. Die Verbindung zur Datenbank 25
1.2.14. Quiz *29*
1.2.15. Zusammenfassung *30*

© Java : Sun Microsystems
© Duke: Sun Microsystems
© mySQL