

Datenströme und Datenbanken

Autor: Dipl.-Math.
E. Engelhardt

Stand: 08. Juni 2009



Inhaltsverzeichnis

- ◆ Streams
- ◆ Exceptions
- ◆ Beispiele für Streams
 - Tastatureingaben
 - Text-Ein- und Ausgaben in Dateien
 - Daten-Ein- und Ausgaben in Dateien
 - Serialisierung von Objekten
- ◆ Client/Server-Anwendung
- ◆ Java und Datenbanken
 - Datenbankbindung mittels JDBC-Treiber
 - SQL

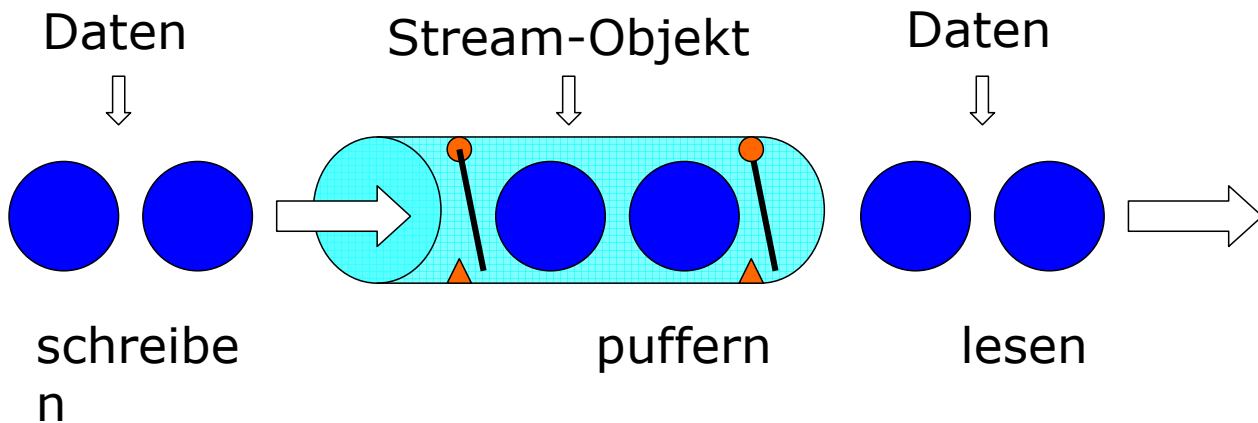
Abkürzungen

- ◆ API Application Programming Interface
- ◆ AWT Abstract Window Toolkit
- ◆ FIFO First In First Out
- ◆ GUI Gaphical User Interface
- ◆ HTTP Hypertext Transport Protocol
- ◆ IO Input Output (Ein- Ausgabeströme)
- ◆ JDBC Java Database Connectivity
- ◆ JVM Java Virtual Machine
- ◆ ODBC Open Database Connectivity
- ◆ RMI Remote Method Interface
- ◆ SQL Structured Query Language
- ◆ URL Uniform Resource Locator

Datenströme und Daten

- ◆ In GUIs (**G**raphical **U**ser **I**nterface) findet die Kommunikation über Botschaften und Ereignisse statt (Messages, Events)
 - Mausbewegungen, Mausklicks
 - Texte in Textfeldern, Auswahlfeldern, ...
 - Menüwahl, Buttons, ...
- ◆ Jede andere Kommunikation mit der Außenwelt (auch dem Benutzer) erfolgt über Datenströme („Streams“)
 - Klassen für Datenströme sind im Paket „java.io“ enthalten
 - Datenströme können mit Rohren verglichen werden
 - Streams arbeiten unidirektional (nur in einer Richtung)
 - sie arbeiten nach dem FIFO-Prinzip (**F**irst **I**n **F**irst **O**ut)
 - Streams fungieren auch als Zwischenpuffer

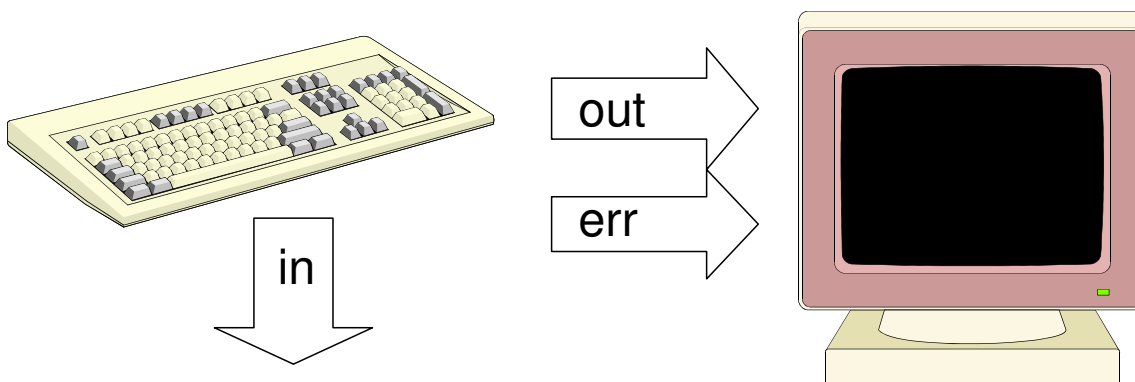
Bildliche Darstellung



- Daten müssen in Streams hineingeschrieben und auch explizit wieder ausgelesen werden
- Die Dateneinheiten können einzelne Bytes, Strings, elementare Datentypen oder serialisierte Objekte sein
- Datenstreams müssen immer als Objekte aus den vorhandenen Klassen gebildet werden

Standard-Streams

- Standard-Streams sind Objekte der byteorientierten Klassen „java.io.InputStream“ und „java.io.OutputStream“
- Sie sind als Konstanten in der Klasse „java.lang.System“ definiert und stehen somit in jedem java-Programm direkt zur Verfügung



- System.in hat Methode read() (erwartet Byte-Array)

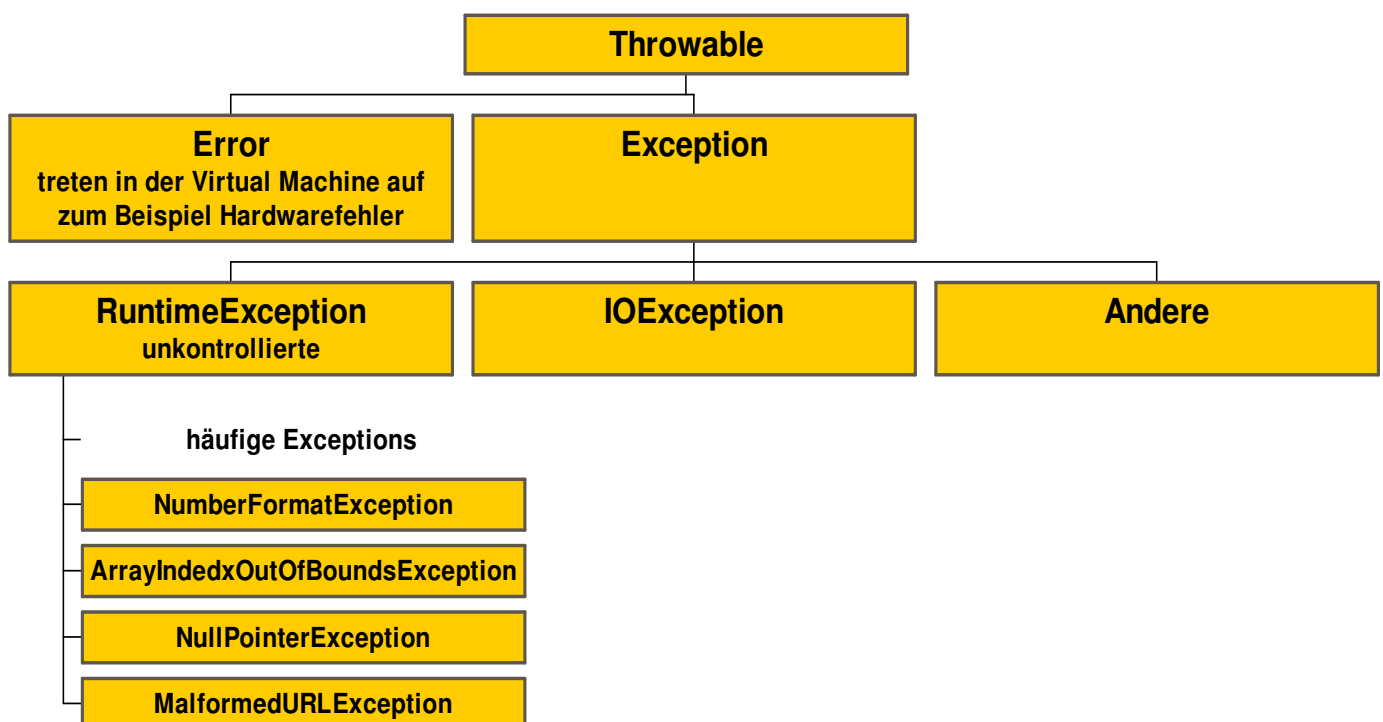
System.out und System.err haben Methoden print() und println()

Exceptions

- ◆ Exceptions sind Ausnahmen, die den normalen Ablauf einer Methode unterbrechen; sie treten nur in Methoden auf
- ◆ Exceptions sind besondere Situationen (auch beheb-bare Fehler)
- ◆ bei bestimmten Anweisungen werden zwingend Exception-Handlings (Ausnahmebehandlungen) erwartet
- ◆ Das Programm kann fortgesetzt werden
- ◆ Durch Errors (Fehler) wird das Programm in der Regel beendet

Exceptions

Klassifizierung der Exceptions



Behandlung von Exceptions

- ◆ In einem „try“-Block werden die Anweisungen, die Exceptions auslösen können, eingeschlossen
- ◆ Es folgen „catch“-Blöcke, die mögliche Exceptions abfangen
- ◆ Es wird maximal ein „catch“-Block abgearbeitet
- ◆ Wenn kein passender „catch“-Block gefunden wurde wird eine eventuell auftretende Exception nicht behandelt!
- ◆ Die „catch“-Blöcke müssen in der Reihenfolge der Ableitung implementiert werden
- ◆ Es kann ein „finally“-Block angegeben werden, der immer ausgeführt wird

Exceptions weiter geben

- ◆ Wird kein passender „catch“-Block gefunden, wird die Exception an die aufrufende Methode weiter gegeben
- ◆ Methoden können über das Schlüsselwort „*throws*“ anzeigen, dass sie eventuell auftretende Exceptions weiter reichen
- ◆ „*RuntimeExceptions*“ sollten nicht weiter gegeben werden (sie sollten gar nicht erst auftreten)
- ◆ wird eine kontrollierte Exception nicht behandelt, wird das Programm nicht compiliert
- ◆ wird eine unkontrollierte Exception nicht behandelt, wird die Standard-Exception-Behandlung ausgeführt

Eigene Exceptions auslösen

- ◆ Erzeugung einer Exception durch:
`throw new xxxException (" ... ");`
- ◆ Es wird eine Nachbehandlung der Fehler durchgeführt
- ◆ Es ist oft effizienter, mit einfachen Tests das Auftreten von Fehlern zu verhindern, als die Fehler mit „try“ und „catch“ zu behandeln
- ◆ Durch Exceptions können aber Fehler im Programm schnell eingegrenzt werden

Tastatureingabe (1)

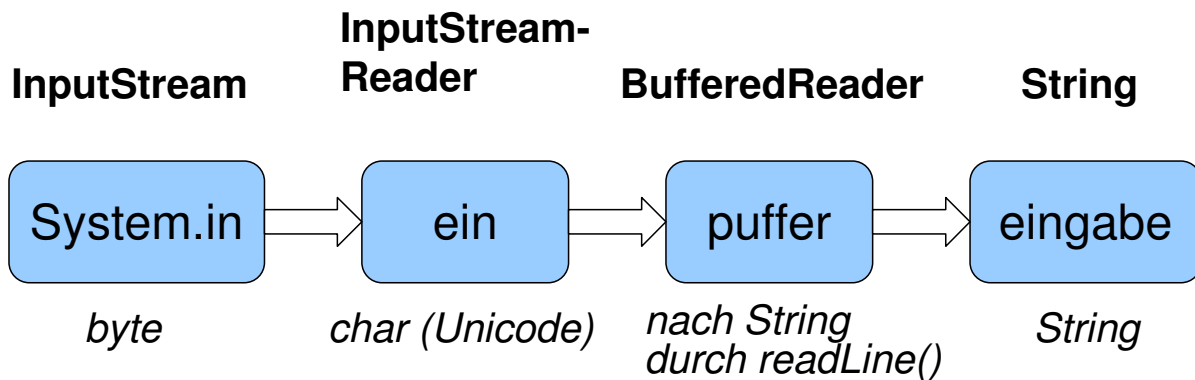
```
import java.io.*;
public class Eingabe1
{
    public static void main(String[] argumente)
    {
        byte puffer[]= new byte[10];
        int anzahl;
        String Eingabe;

        System.out.print("Eingabe: ");
        try
        {
            anzahl= System.in.read(puffer, 0, 10);
            Eingabe= new String(puffer, 0, anzahl);
            System.out.println(Eingabe);
        }
        catch (IOException e)
        {
            System.err.println("Fehler: " + e.getMessage());
        }
    }
}
```

- ◆ Wenn mehr als zehn Zeichen eingegeben werden, werden diese nicht gespeichert

Tastatureingabe (2)

- ◆ Die Bytes des byte-Array werden in UnicodeZeichen konvertiert und in ein String-Objekt umgewandelt
- ◆ Der Umweg über ein byte-Array kann über verknüpfte Eingabeströme vermieden werden



- ◆ Anzahl der Zeichen muss nicht zwischen gespeichert werden
- ◆ String muss nicht durch Konstruktor erzeugt werden

Tastatureingabe (3)

```
...
String Eingabe;
InputStreamReader Ein= new InputStreamReader(System.in);
BufferedReader Tastatur= new BufferedReader(Ein);

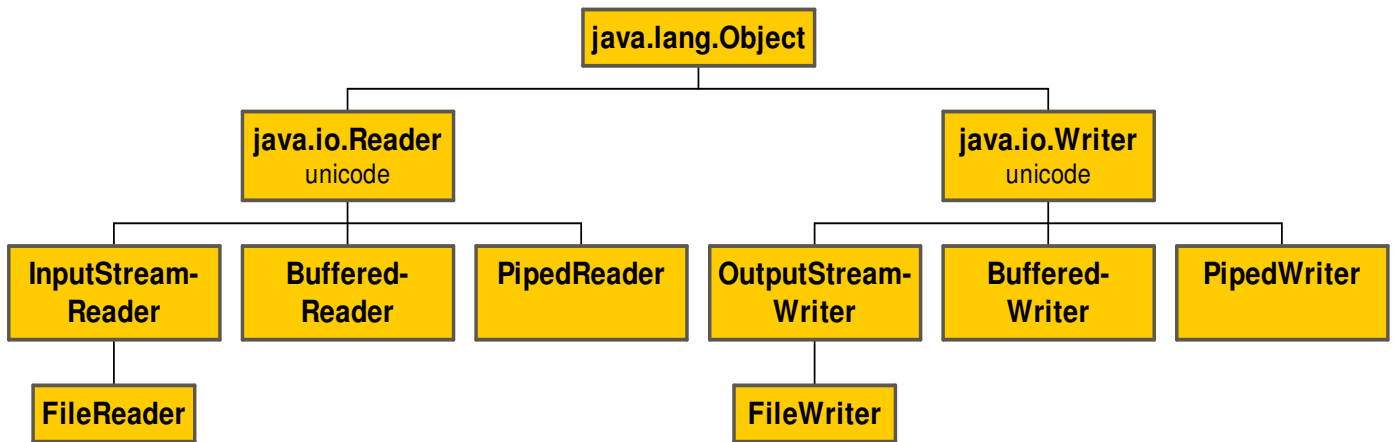
System.out.print("Eingabe: ");

try
{
    Eingabe= Tastatur.readLine();
    System.out.println(Eingabe);
}
catch (IOException e)
{
    System.err.println("Fehler: " + e.getMessage());
}
...
```

- ◆ Ausnahmebehandlung wird erzwungen
- ◆ Durch Plattformunabhängigkeit ist der Umweg über byte-Puffer begründet

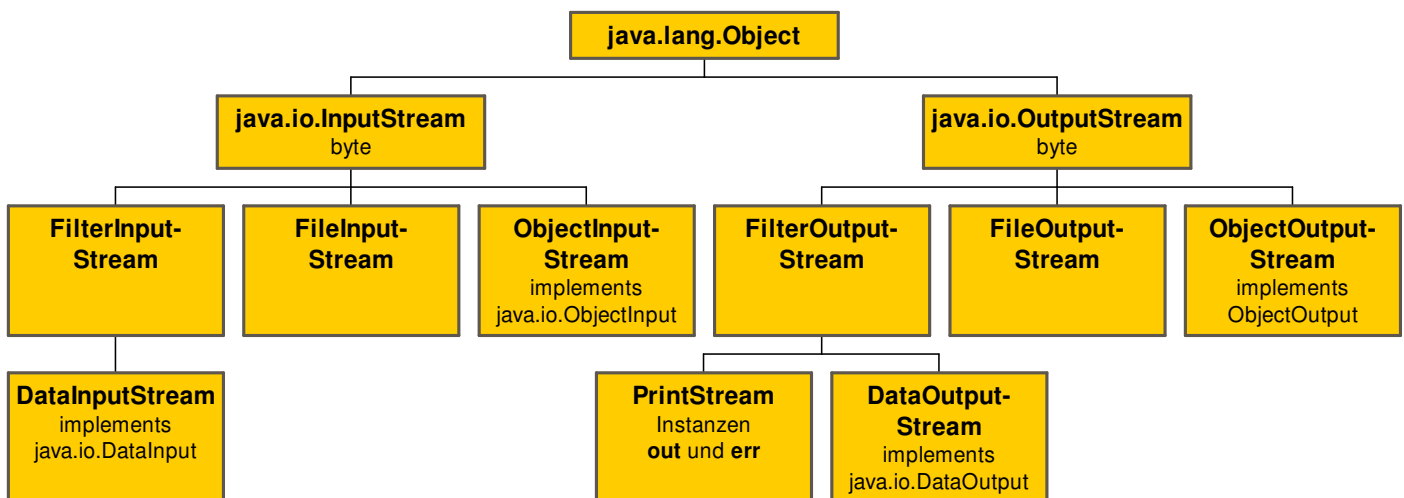
Hierarchie der IO-Klassen (1)

IO-Klassen-Hierarchie unicode-orientiert



Hierarchie der IO-Klassen (2)

IO-Klassen-Hierarchie byte-orientiert



Text lesen (1)

```
...
String Textzeile;
try
{
    FileReader Datei= new FileReader("Beispiel.txt");
    BufferedReader Ein= new BufferedReader(Datei);
    Textzeile= Ein.readLine();           // Vorlesen aus Datei

    while (Textzeile != null)           // Null-Referenz
    {
        System.out.println(Textzeile);
        Textzeile= Ein.readLine();      // Nachlesen aus Datei
    }
    Ein.close();                         // Schließen
}
catch (FileNotFoundException e)
{
    System.err.println("Datei nicht gefunden");
}
catch (IOException e)
{
    System.err.println("Fehler: " + e.getMessage());
}
...
```

Text schreiben (2)

```
...
String Textzeile;
try
{
    FileWriter Datei= new FileWriter("Beispiel2.txt");
    BufferedWriter Aus= new BufferedWriter(Datei);
    InputStreamReader Ein= new InputStreamReader(System.in);
    BufferedReader Tastatur= new BufferedReader(Ein);

    System.out.print("Text eingeben: ");
    Textzeile= Tastatur.readLine();      // Vorlesen Tastatur

    while (Textzeile.compareTo("#") != 0) // Stringvergleich
    {
        Aus.write(Textzeile);
        Aus.newLine();                   // systemabhängiges NL
        System.out.print("Text eingeben: ");
        Textzeile= Tastatur.readLine();  // Nachlesen Tastatur
    }
    Aus.close();                         // Schließen
}
catch (IOException e)
{
    System.err.println("Fehler: " + e.getMessage());
}
...
```

Textdateien (3)

- ◆ Das „Vorlesen“ ist notwendig, da erst nach einem Leseversuch festgestellt werden kann, ob das Dateiende erreicht ist
- ◆ `FileWriter("Beispiel2.txt")` erstellt neue Datei
- ◆ `FileWriter("Beispiel2.txt", true)` erweitert bestehende Datei, sofern vorhanden
- ◆ `BufferedReader` bzw. `BufferedWriter` sorgen dafür, dass nicht jedes Byte einzeln gelesen bzw. geschrieben wird
- ◆ `aus.newLine()`; schreibt ein systemabhängiges Zeilenende
- ◆ Daten werden erst nach `Datei.close()`; auf die Platte geschrieben

Datendateien (1)

- ◆ Textdateien sind geeignet, wenn die Daten mit einem beliebigen Editor lesbar sein sollen
- ◆ Für viele Daten (z.B. Gleitkommazahlen) müsste immer eine Konvertierung in String und nach dem Lesen zurück erfolgen
- ◆ „`DataInputStream`“ und „`DataOutputStream`“ lesen bzw. schreiben elementare Datentypen
- ◆ Diese Streams werden mit „`FileInputStream`“ und „`FileOutputStream`“ verknüpft, um Daten auf Dateien zu schreiben bzw. von Dateien zu lesen
- ◆ Die Daten liegen in der Datei in einem durch das Programm direkt lesbarem Format vor
- ◆ Solche Dateien sollten nur mit „NotePad“ oder einem Hex-Editor außerhalb des Programms geöffnet werden

Daten schreiben (2)

```
...
double temperatur;

try
{
    FileOutputStream Datei= new FileOutputStream("Beispiel.dat");
    DataOutputStream Aus= new DataOutputStream(Datei);
    InputStreamReader Ein= new InputStreamReader(System.in);
    BufferedReader Tastatur= new BufferedReader(Ein);

    // Vorlesen der Tastatureingabe
    System.out.print("Temperatur eingeben: ");
    temperatur= Double.parseDouble(Tastatur.readLine());

    while (temperatur != -9999)
    {
        Aus.writeDouble(temperatur);
    // Nachlesen der Tastatureingabe
        System.out.print("Temperatur eingeben: ");
        temperatur= Double.valueOf(Tastatur.readLine()).doubleValue();
    }
    Aus.close();    // Schließen des Ausgabe-Streams
}
catch (IOException e)
{
    System.err.println("Fehler: " + e.getMessage());
    e.printStackTrace();
}
...
```

Daten lesen (3)

```
...
double temperatur;
DataInputStream Ein= null;

try
{
    FileInputStream Datei= new FileInputStream("Beispiel.dat");
    Ein= new DataInputStream(Datei);

    while (true)
    {
        temperatur= Ein.readDouble();
        System.out.println("" + temperatur);
    }
}
catch (EOFException e)
{
    System.err.println("Dateiende");
}
catch (IOException e)
{
    System.err.println("Fehler: " + e.getMessage());
}
try
{
    Ein.close();    // Schließen des Eingabe-Streams
}
catch (IOException e) {}
...
```

Datendateien (4)

- ◆ Beim Lesen des Dateiende von Datendateien wird eine Exception geworfen.
- ◆ Diese muss in einem catch-Zweig abgefangen werden (der Block kann leer sein)
- ◆ Es gibt keine Methode zum Erstellen von Dateien.
- ◆ Dateien werden durch die entsprechenden Streams bei Bedarf erzeugt
- ◆ In Java steht ein Paket für einige weit verbreitete Komprimieralgorithmen zur Verfügung
- ◆ Natürlich gibt es auch Klassen für wahlfreien Zugriff

Objekte serialisieren und laden (1)

- ◆ Zur Speicherung von Objekten stehen die Streams „ObjectInputStream“ und „ObjectOutputStream“ zur Verfügung
- ◆ Serialisieren bedeutet, dass die Daten des Objektes in einen byte-Strom umgewandelt werden (umgekehrt durch deserialisieren)
- ◆ Dazu stehen die Methoden „writeObject“ und „readObject“ zur Verfügung
- ◆ Objekte sind nicht automatisch serialisierbar. In der Klasse muss das Flag-Interface „Serializable“ implementiert werden
- ◆ Attribute, deren Werte nicht gespeichert werden sollen, werden mit „transient“ gekennzeichnet
- ◆ Untergeordnete Objekte werden auch serialisiert (rekursiv)
- ◆ Es wird aber jedes Objekt nur einmal serialisiert.

Objekte serialisieren (2)

```
public class Auto implements Serializable
...
Auto Trabbi= Auto(26);
try
{
    FileOutputStream Datei= new
    FileOutputStream("autos.dat");
    ObjectOutputStream Aus= new ObjectOutputStream(Datei);

    Aus.writeObject(Trabbi);

    Aus.close();    // Schließen des Ausgabe-Streams
}
catch (IOException e)
{
    e.printStackTrace();
}
...
```

- ◆ Es wird nur ein Objekt gespeichert
- ◆ Die Daten sind nicht sinnvoll mit einem Texteditor lesbar

Objekte laden (3)

```
...
Auto neuer_Trabbi;
try
{
    FileInputStream Datei= new FileInputStream("autos.dat");
    ObjectInputStream Ein= new ObjectInputStream(Datei);

    neuer_Trabbi= (Auto) Ein.readObject();

    Ein.close();    // Schließen
}
catch (IOException e)
{
    e.printStackTrace();
}
...
```

- ◆ „readObject“ liefert einen Verweis auf ein Objekt der Klasse „Object“, deshalb muss die Zuweisung gecastet werden

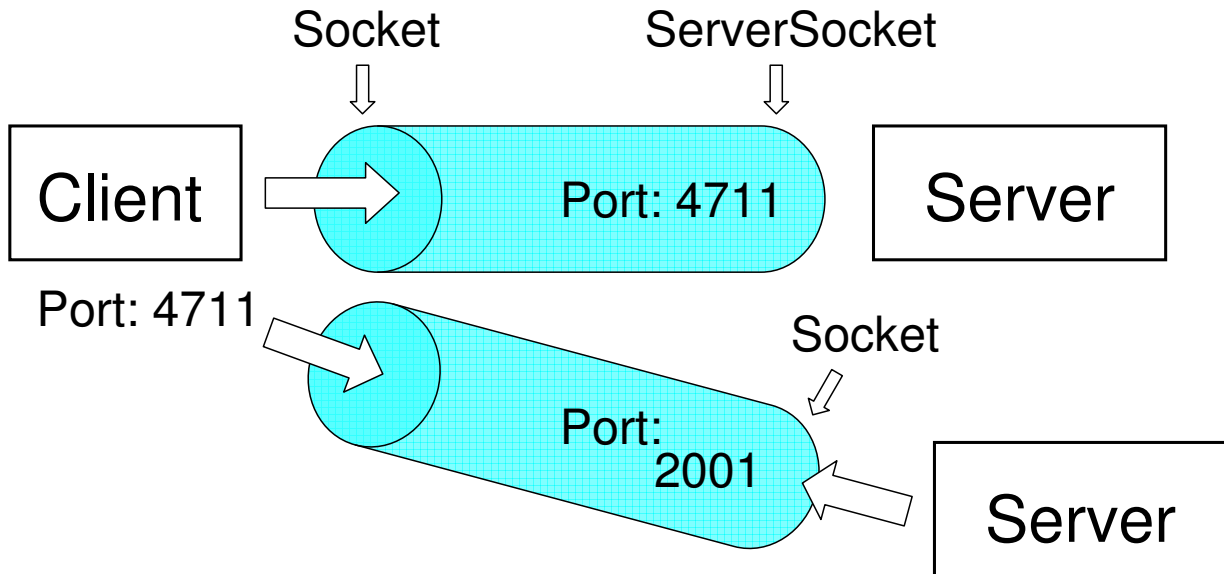
Stufen der Client-Server-Anwendung

- ◆ Übersicht
- ◆ Grundgerüst des Servers
 - Stufe 1: Horcher-Socket anlegen
 - Stufe 2: Warten auf Client und Verbindungs-Socket anlegen, Ein- Ausgabe-Objekte anlegen, String-Objekt anlegen
 - Stufe 3: while-Schleife für Verbindung
- ◆ Grundgerüst des Client
 - Stufe 4: Verbindungs-Socket anlegen
 - Stufe 5: Ein- Ausgabe-Objekte anlegen, String-Objekte anlegen
 - Stufe 6: while-Schleife für Verbindung

Sockets

- ◆ Die Rechner werden über die IP-Nummern identifiziert
- ◆ Kommunikationskanäle repräsentieren die Verbindung zweier Ports
- ◆ die Endknoten der Kommunikationsverbindungen sind die Sockets (IP-Adresse + Port-Nummer)
 - Es lassen sich 65.535 Ports ansprechen
 - Die Port-Nummern kleiner 1024 sind für Systemdienste reserviert (telnet: 23, ftp: 20/21, smtp: 25, login: 513)
- ◆ für die Verbindungsaufnahme müssen dem Client die IP-Nummer und die Port-Nummer bekannt sein
- ◆ Zuerst werden Server und Client auf einem Rechner ausprobiert (IP-Adresse: 127.0.0.1; localhost)

Verbindungsaufnahme



- ◆ Nach Verbindungsaufnahme durch den Client liefert der Server ein neues Socket für die Verbindung

Grundgerüst des Servers

```
import java.net.*;
import java.io.*;

public class Server
{
    public static void main(String Argumente[])
    {
        BufferedReader NetzEingabe = null;
        PrintStream NetzAusgabe = null;

        1 // Programmteil

    }
}
```

Stufe 1: HorcherSocket

1

```
try
{
    ServerSocket HorchSocket = new ServerSocket(4711);

    2 // VerbindungsSocket, Warten auf Client
    // Ein- Ausgabe-Objekte und Strings

    3 // while-Schleife zur Verbindung
}
catch (Exception e)
{
    System.err.println("Fehler! " + e);
}
```

Stufe 2: VerbindungsSocket, E/A-Objekte

2

```
// Horchen, bis sich Client meldet
System.out.println("Server wartet auf Client");

Socket VerbindungsSocket = HorchSocket.accept();
System.out.println("Client hat sich eingewählt!");

NetzAusgabe = new
    PrintStream(VerbindungsSocket.getOutputStream());
NetzEingabe = new BufferedReader(new
    InputStreamReader(VerbindungsSocket.
        getInputStream()));

String EmpfangZeile = new String();
```

Stufe 3: while-Schleife für Verbindung

3

```
// Nach Empfang von "QUIT" wird Server beendet

while (!EmpfangZeile.equals("QUIT"))
{
    EmpfangZeile = NetzEingabe.readLine();
    Netzausgabe.println(" -> " + EmpfangZeile);
    System.out.println("Client: " + EmpfangZeile);
}
System.exit(0);
```

Grundgerüst des Client

```
import java.net.*;
import java.io.*;

public class Client
{
    public static void main(String Argumente[])
    {
        BufferedReader NetzEingabe = null;
        BufferedReader Tastatur = null;
        PrintStream Netzausgabe = null;

        4 // Programmteil

    }
}
```

Stufe 4: VerbindungsSocket

4

```
try
{
    Socket VerbindungsSocket =
        new Socket("127.0.0.1", 4711);

    5 // Ein- Ausgabe-Objekte und Strings
    6 // while-Schleife zur Verbindung
}
catch (Exception e)
{
    System.err.println("Fehler! " + e);
}
```

Stufe 5: E/A-Objekte, Strings

5

```
NetzEingabe = new BufferedReader(new
    InputStreamReader(VerbindungsSocket.
        getInputStream()));
Tastatur = new BufferedReader(new
    InputStreamReader(System.in));
Netzausgabe = new
    PrintStream(VerbindungsSocket.getOutputStream());

System.out.println("Verbindung ist aufgebaut!");

String EmpfangZeile = new String();
String SendeZeile = new String();
```

Stufe 6: while-Schleife für Verbindung

6

```
while (!SendeZeile.equals("QUIT"))
{
    System.out.println(
        "Eingabe (QUIT beendet den Client): ");
    SendZeile = Tastatur.readLine();
    Netzausgabe.println(SendeZeile);
    EmpfangZeile = NetzEingabe.readLine();
    System.out.println("Echo: " + EmpfangZeile);
}

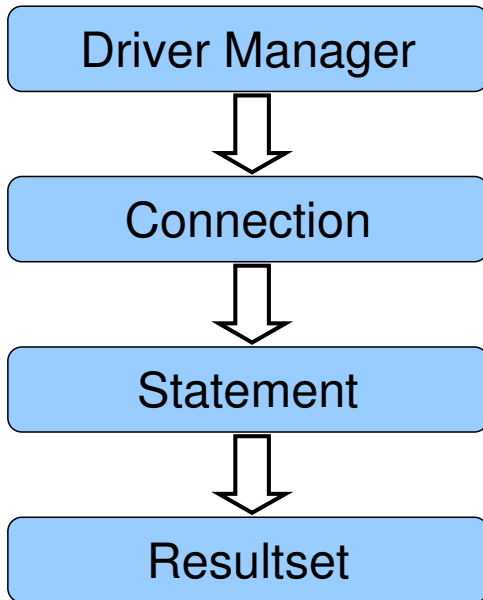
System.exit(0);
```

Java und Datenbanken

- ◆ Datenbankanbindung unter Java
- ◆ Stufen des SQL-Test
 - Grundgerüst: keine Wirkung
 - Stufe 1: ODBC-JDBC-Treiber laden
 - Stufe 2: Verbindung zur Datenbank herstellen
 - Stufe 3: Zugriff auf Datenbank durch „Statement“
 - Stufe 4: Ergebnis einer SQL-Abfrage in „ResultSet“ speichern
 - Stufe 5: Ergebnis auf Monitor anzeigen
- ◆ SQL-Exkurs

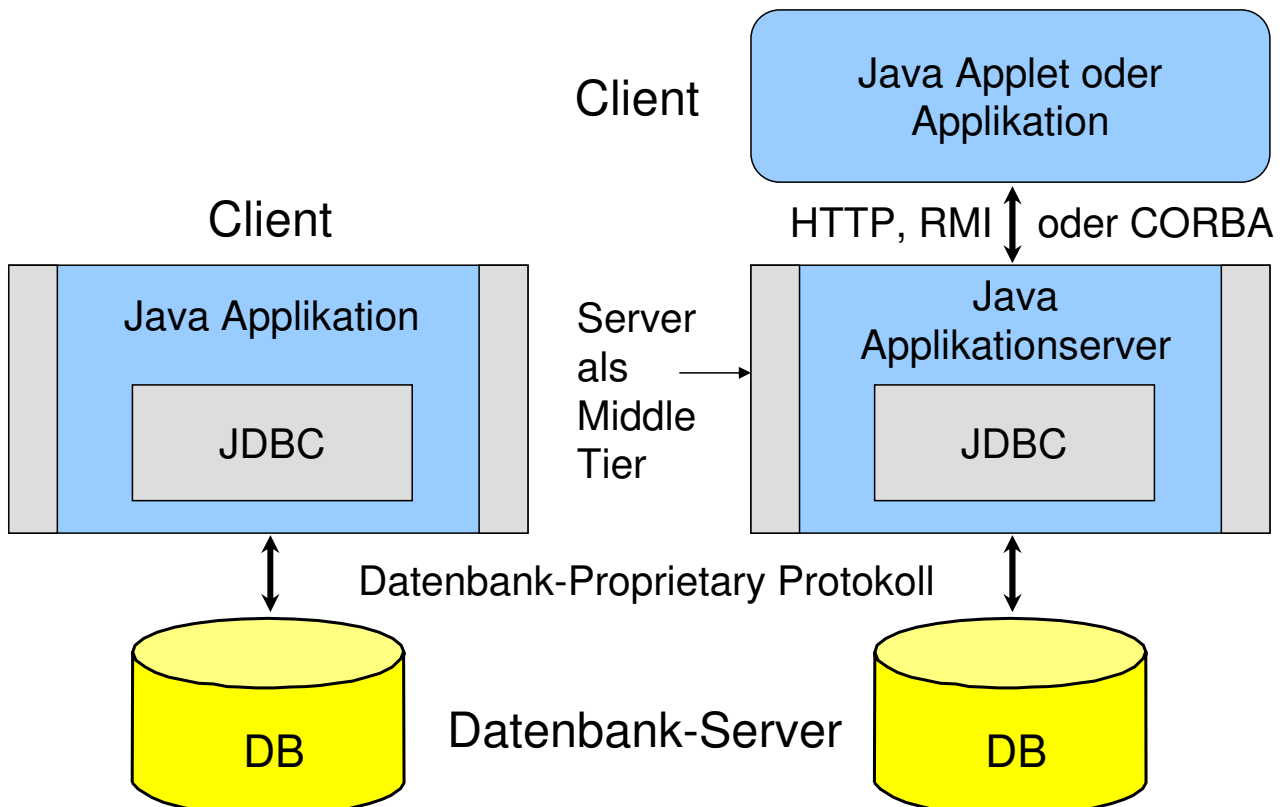
Datenbankanbindung

- ◆ JDBC API stehen unter „*java.sql*“ und „*javax.sql*“ zur Verfügung
- ◆ Der Zugriff erfolgt über die folgenden Stufen:



- Low-Level-Schicht des JDBC; setzt auf herstellerabhängigen Treiber auf und stellt Zugriff bereit
- stellt Verbindung über den Treiber zur Datenbank her
- Aufbau der SQL-Anweisung und Weiterleitung an Datenbank
- nimmt Ergebnisse der Datenbankabfrage entgegen

Two-Tier- und Three-Tier-Model



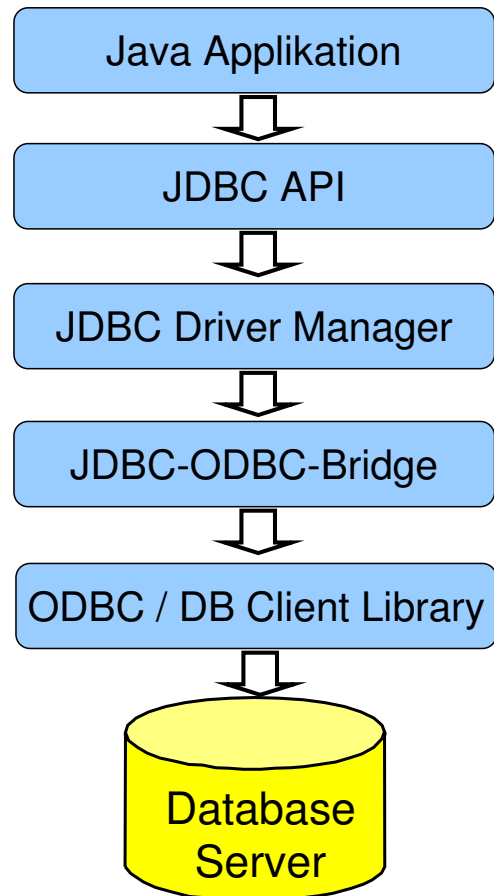
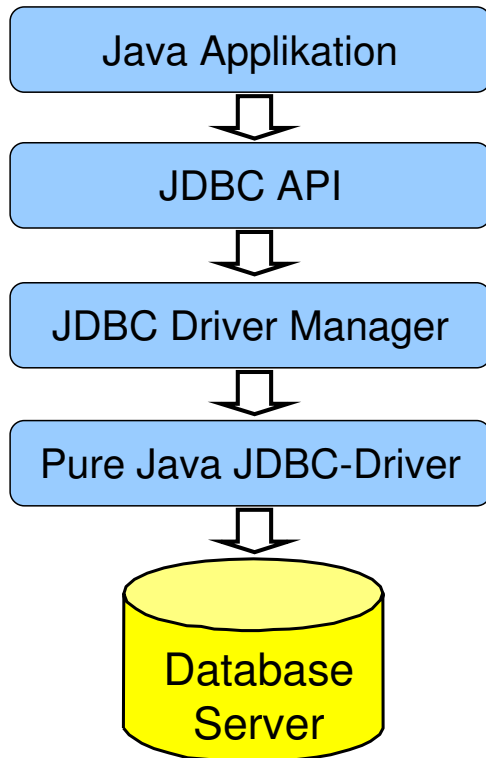
JDBC-Treiber (1/3)

- ◆ 1. JDBC-ODBC Bridge
 - einfacher Zugriff über den Standard-ODBC-Treiber (Open Database Connectivity)
 - muss auf jedem Client-Rechner installiert sein
- ◆ 2. JDBC-Native Treiber
 - „Native-API-partly Java Technology-enabled Driver“ greift auf proprietären Treiber des Datenbankherstellers zu
 - muss auf jedem Client-Rechner installiert sein
- ◆ 3. JDBC-Netzwerk-Treiber
 - „Net-protocol fully Java Technology-enabled Driver“
 - reine Java-Lösung auf dem Client
 - JDBC-API-Aufrufe werden in DBMS-unabhängiges Netzwerkprotokoll übersetzt und dem DBMS-eigenen Protokoll des Datenbank-Servers übergeben (Drei-Schichten-Lösung; Three-Tier-Model)

JDBC-Treiber (2/3)

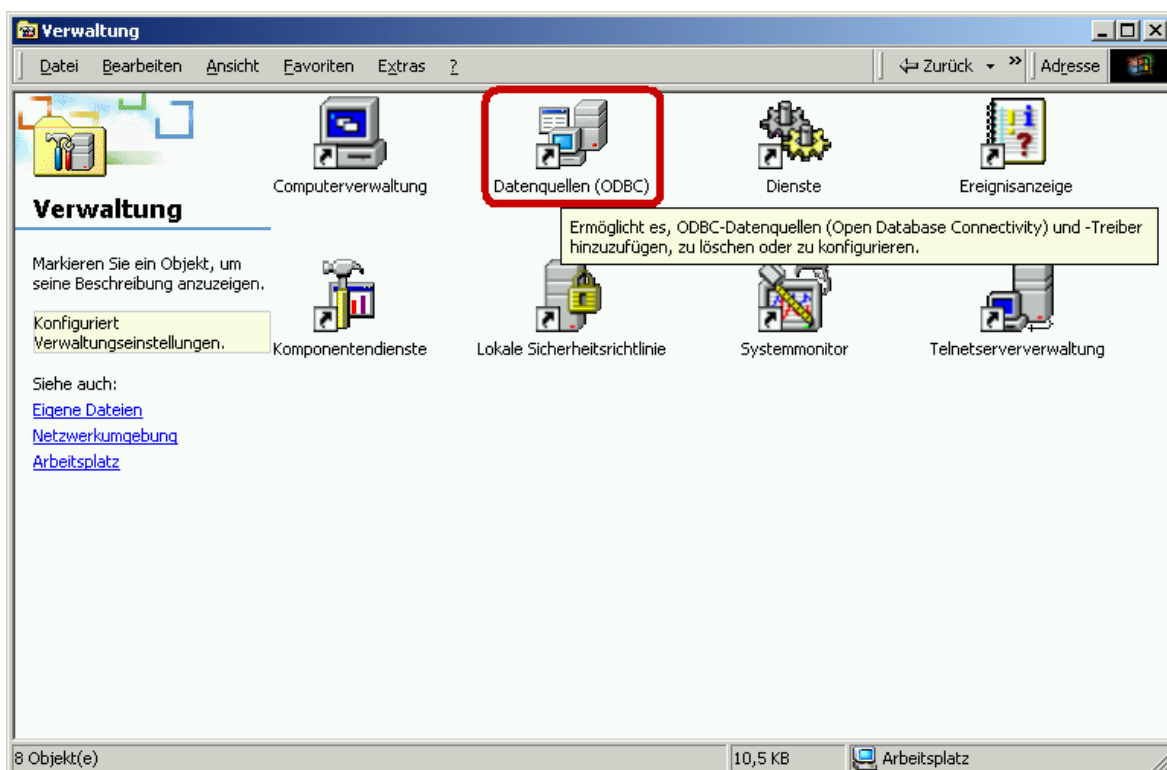
- ◆ 4. Direktzugriff
 - „Native-protocol fully Java Technology-enabled Driver“
 - JDBC-API-Aufrufe werden direkt in das DBMS-eigene Protokoll des Datenbank-Servers konvertiert
- ◆ JDBC-Wrapper-Klasse
 - Unterschiede der Datenbanksysteme werden abgefangen und das Verhalten wird angepasst

JDBC-Treiber (3/3)



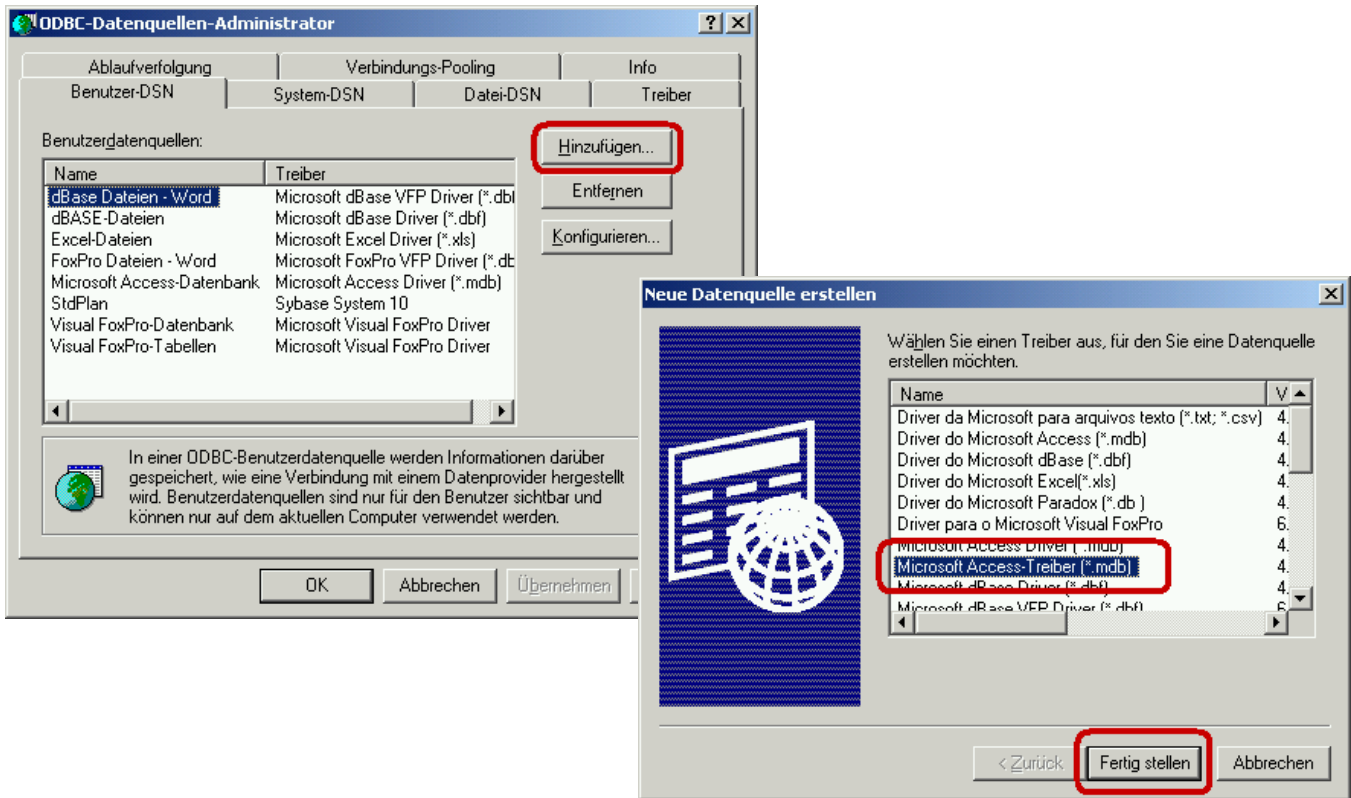
ODBC-Datenbankquelle einrichten

- ◆ Windows XP-Systemsteuerung → Verwaltung



ODBC-Datenbankquelle einrichten

- ◆ Windows XP-Systemsteuerung → Verwaltung

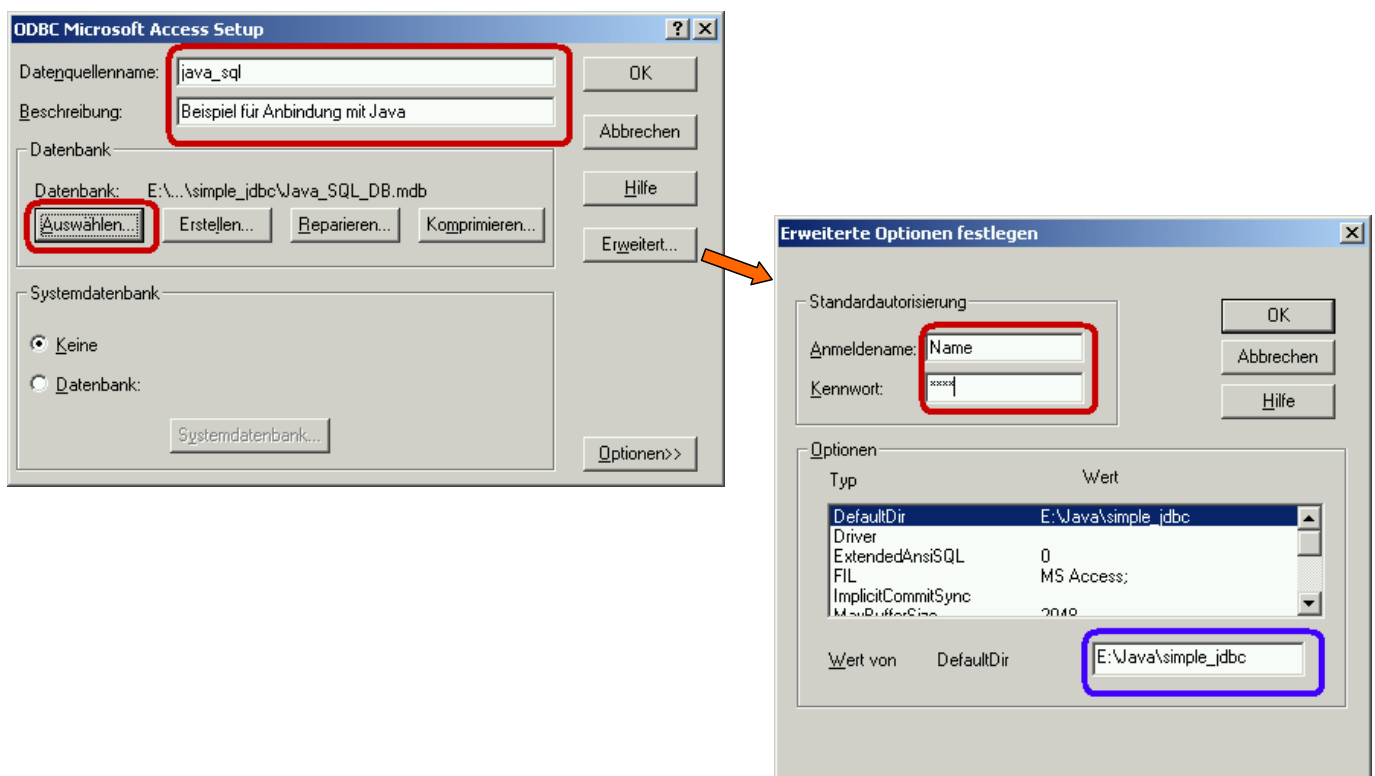


Folie 45 von 55

Datenströme und Datenbanken

ODBC-Datenbankquelle einrichten

- ◆ Windows XP-Systemsteuerung → Verwaltung

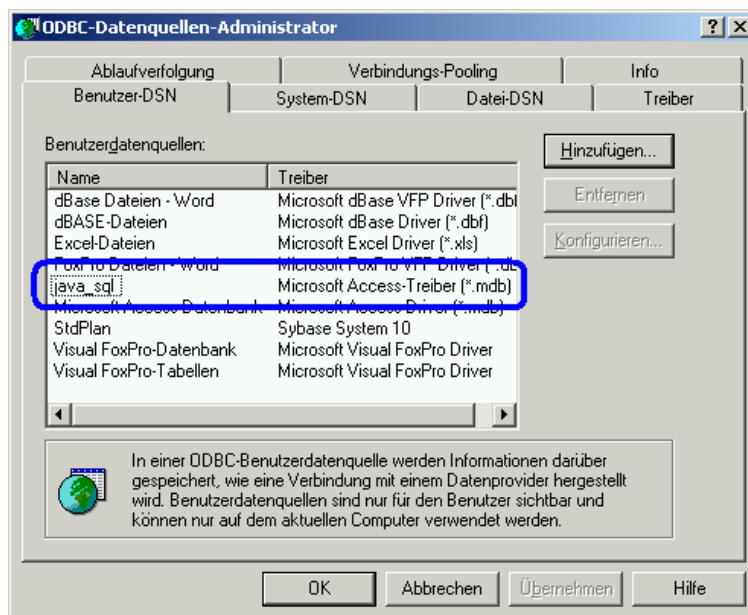


Folie 46 von 55

Datenströme und Datenbanken

ODBC-Datenbankquelle einrichten

- ◆ Windows XP-Systemsteuerung → Verwaltung



Folie 47 von 55

Datenströme und Datenbanken

Grundgerüst SQL_Test (1/5)

```
import java.sql.*;

public class SQL_Test
{
    public static void main(String Argumente[])
    {
        try
        {
            1a 2a 3a 4a
            5 // Ergebnis-Anzeige
            3b 2b // Schließen
        }
        1b 2c 3c 4b // catch-Zweige
    }
}
```

Folie 48 von 55

Datenströme und Datenbanken

Stufe 1: ODBC-JDBC-Treiber laden (2/5)

1a

```
// Laden des ODBC-Treibers
// muss fehlerfrei auch ohne DB funktionieren
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

1b

```
catch (ClassNotFoundException ex)
{
    System.err.println("Treiber nicht gefunden!");
}
```

Stufe 2: Verbindung zur Datenbank (3/5)

2a

```
Connection db = DriverManager.getConnection(
    "jdbc:odbc:test", "Name", "Passwort");
```

Name der ODBC-Datenquelle

Anmeldename

Kennwort

2c

3c

4b

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

Stufe 3 und 4: Zugriff auf Datenbank (4/5)

```
Statement stmt = db.createStatement();
```

3a

```
ResultSet rs = stmt.executeQuery(  
    "SELECT * FROM adressen");
```

4a

```
while (rs.next())
```

5

```
{
```

```
    System.out.println(rs.getString("Name") +  
        ": " + rs.getString("Strasse") +  
        ": " + rs.getString("TelNr"));
```

```
}
```

```
stmt.close();
```

3b

2b

```
db.close();
```

Stufe 5: Ergebnis der SQL-Abfrage anzeigen (5/5)

5

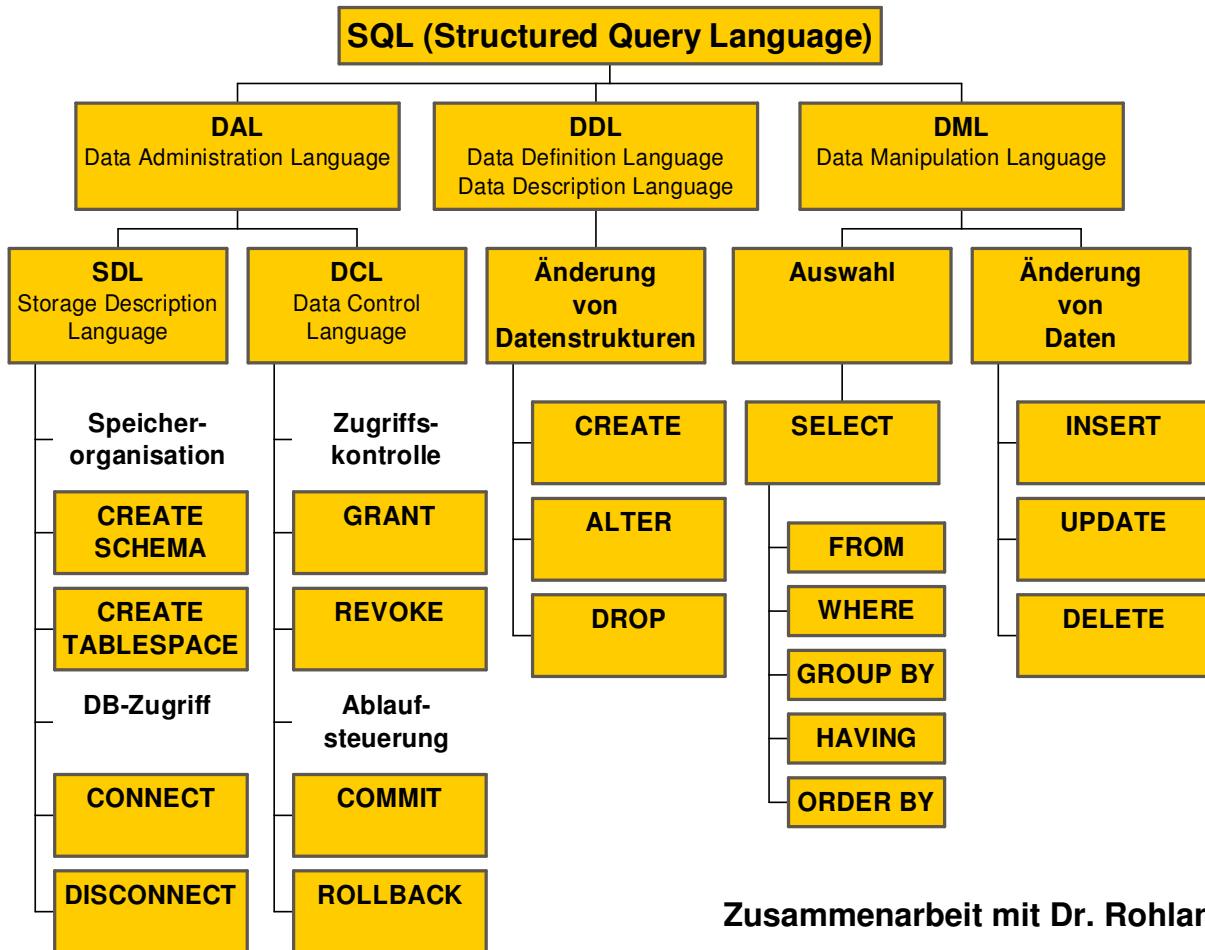
```
while (rs.next())
```

```
{
```

```
    System.out.println(rs.getString("Name") +  
        ": " + rs.getString("Strasse") +  
        ": " + rs.getString("TelNr"));
```

```
}
```

Bestandteile der SQL



Folie 53 von 55

Datenströme und Datenbanken

SELECT-Abfrage

SELECT (*Feldname, Feldname, ... | **)

FROM *Tabelle [, Tabelle, Tabelle, ...]*

[WHERE (*Bedingung*)] (Bsp.: name LIKE `D%`)

[GROUP BY *Feldname* [HAVING (*Bedingung*)]]

[ORDER BY *Feldname* [ASC | DESC] ...]

Folie 54 von 55

Datenströme und Datenbanken

Literaturangaben

- [1] Elke Niedermair / Michael Niedermair:
Internet-Programmierung mit Java,
Data Becker GmbH & Co. KG,
ISBN 3-8158-2086-3
- [2] Alexander Niemann: *Das Einsteigerseminar*,
Objektorientierte Programmierung in Java
bhv Verlag Bürohandels- und Verlagsgesellschaft mbH,
ISBN 3-8287-1015-8
- [3] Java 2 SDK v 1.2.2, *Grundlagen Programmierung*
HERDT-Verlag für Bildungsmedien GmbH, Nackenheim