BERUFSAKADEMIE SACHSEN
STAATLICHE STUDIENAKADEMIE
DRESDEN
UNIVERSITY OF COOPERATIVE EDUCATION

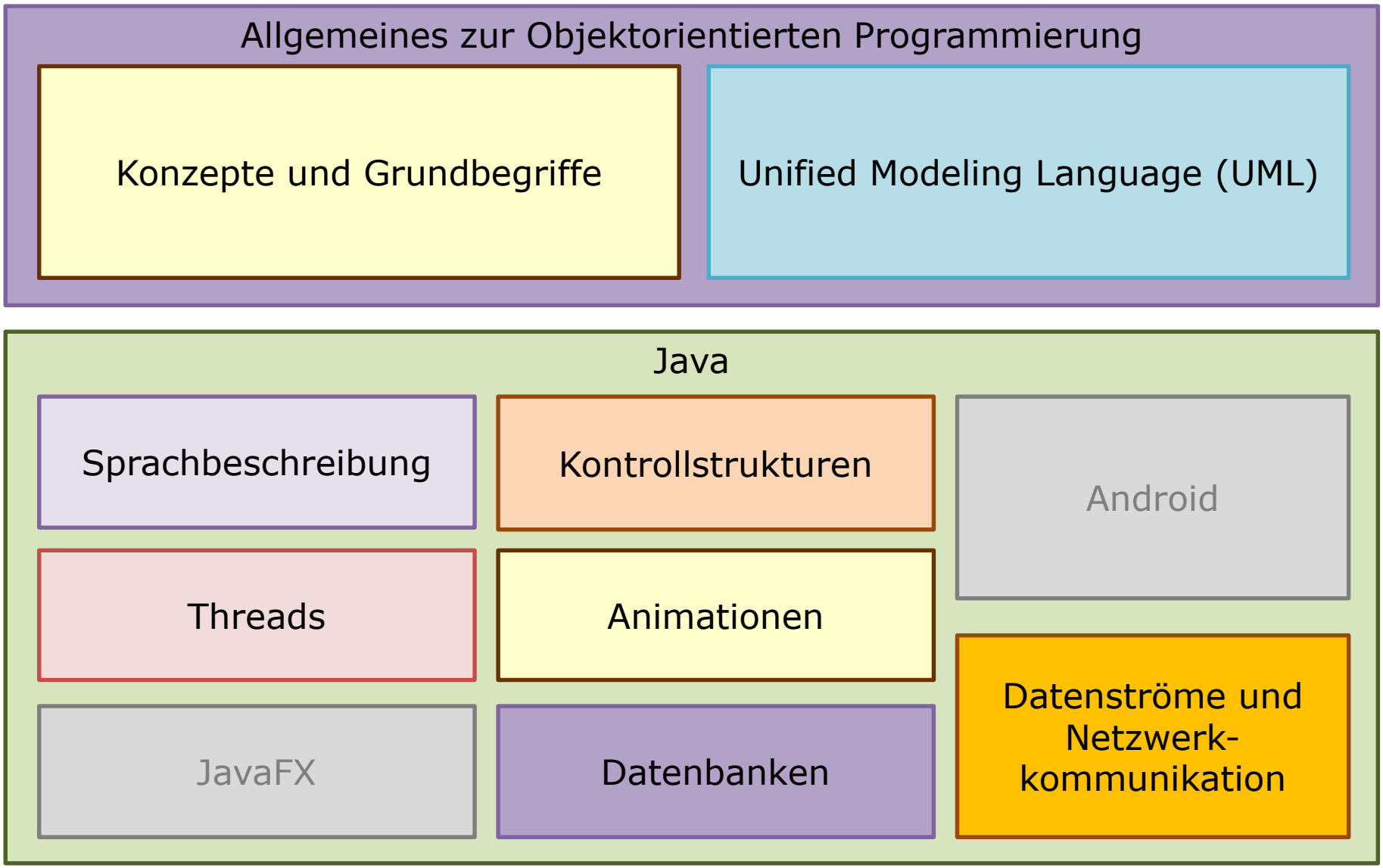
Objektorientierte Programmierung

Datenströme, Netzwerke und Datenbanken

mit Material von Prof. E. Engelhardt

Prof. Dr.-Ing. Tenshi Hara
tenshi.hara@ba-sachsen.de

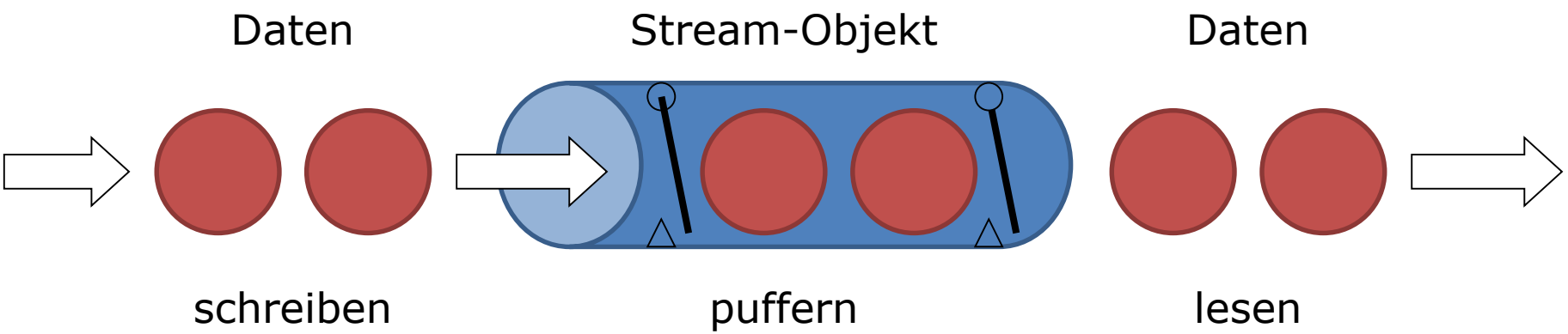
Aufbau der Lehrveranstaltung



Datenströme und Daten

- in grafischen Nutzungsoberflächen (Graphical User Interface; GUI) findet Kommunikation mit Botschaften und Ereignisse statt (Messages, Events)
  - Mausbewegungen, Mausklicks, …
  - Textfelder, Auswahlfelder, …
  - Menüwahl, Schaltflächen (Button), …
- jede andere Kommunikation (auch mit Nutzer): Datenströme (Streams)
  - Klassen für Datenströme sind im Paket `java.io` enthalten
  - Datenströme können mit Rohren verglichen werden
  - Streams arbeiten unidirektional
  - Sie arbeiten nach dem FIFO-Prinzip
  - Streams fungieren auch als Zwischenpuffer (Buffer Stream)

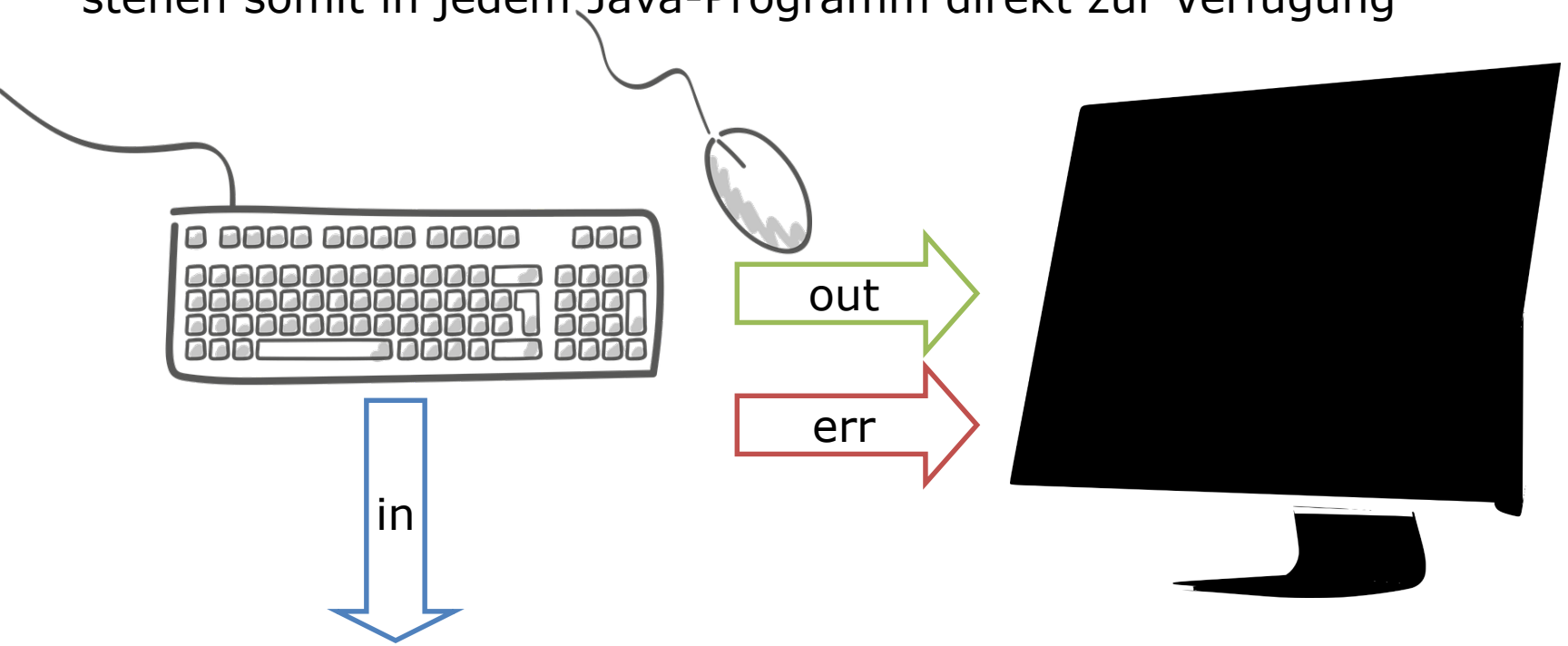
Bildliche Darstellung



- Daten müssen in Streams geschrieben und *explizit* gelesen werden
- Dateneinheiten können einzelne Bytes, Strings, elementare Datentypen oder serialisierte Objekte sein
- Streams werden immer als Objekte aus vorhandenen Klassen gebildet

Standard-Streams

- Standard-Streams sind Objekte der Byte-orientierten Klassen `java.io.InputStream` und `java.io.PrintStream`
- Sie sind als Konstanten in der Klasse `java.lang.System` definiert und stehen somit in jedem Java-Programm direkt zur Verfügung



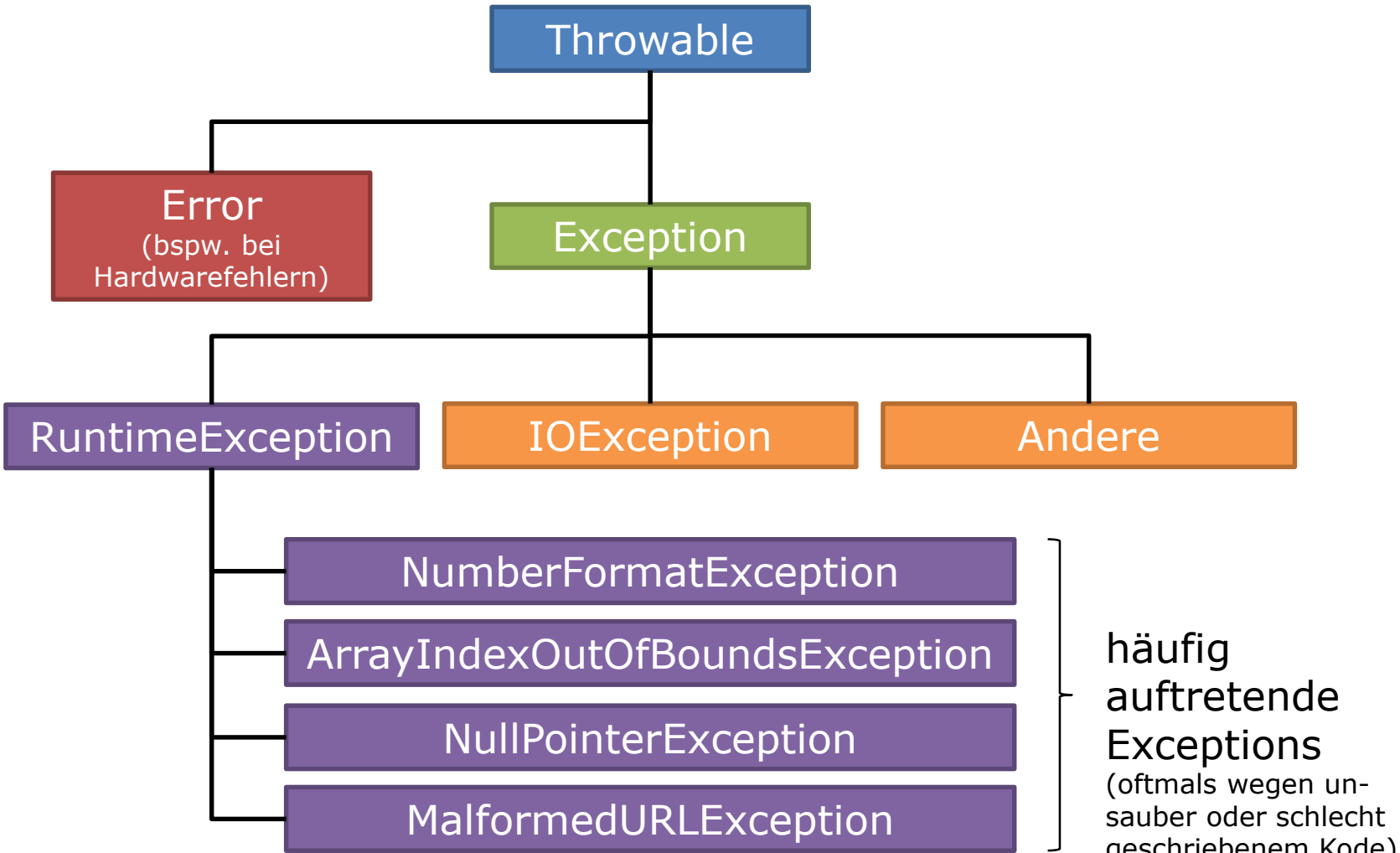
- `System.in` hat Methode `read()` (erwartet Byte-Array)
- `System.out` und `System.err` haben Methoden `print()` und `println()`

Ausnahmesituationen: Exceptions

Exceptions

- Ausnahmen, die den normalen Ablauf einer Methode unterbrechen
- treten nur in Methoden auf
- sind besondere Situationen (behandelbare, ggf. behebbare Fehler)
- bestimmte Anweisungen erwarten zwingend Ausnahmebehandlungen (Exception Handlings)
- Programm kann i.d.R. fortgesetzt werden
- durch Fehler (Error) wird das Programm in der Regel beendet
  → Exceptions sind Vorstufe zum Error

Klassifizierung der Exceptions



Behandlung von Exceptions

- Anweisungen, die Exceptions auslösen können, werden in `try`-Block gekapselt (sie werden „ausprobiert“)
- auftretende Exceptions werden in `catch`-Blöcken abfangen (behandelt)
  - ohne Exception werden alle `catch`-Blöcke übersprungen
  - es wird maximal ein `catch`-Block abgearbeitet
  - ohne passenden `catch`-Block kann Exception nicht behandelt werden! (Typisierung (bspw. vorherige Folie) beachten!)
  - `catch`-Blöcke müssen in der Reihenfolge der Ableitung implementiert werden
- es kann ein `finally`-Block angegeben werden, der immer ausgeführt wird („schlussendlich nach allem anderen“)

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 („werfen“)
- `RuntimeExceptions` sollten nicht weiter gegeben werden
  (sie sollten gar nicht erst auftreten → Indiz für schlechten Kode)
- Programm kann bei definierten, aber unbehandelten Exceptions (kontrollierte Ausnahme) nicht kompilieren
- bei undefinierten Exceptions wird die Standard-Exception-Behandlung ausgeführt (unkontrollierte Ausnahme) und das Programm kompiliert

Eigene Exceptions auslösen

- Erzeugung einer Exception durch

```
throw new xxxException("…");
```

- es wird eine Nachbehandlung der Fehler durchgeführt
- oft effizienter, mit einfachen Tests das Auftreten von Fehlern zu verhindern, als die Fehler `try…catch` zu behandeln
- aber: durch Exceptions können Fehler schnell eingegrenzt werden

Text und Dateien

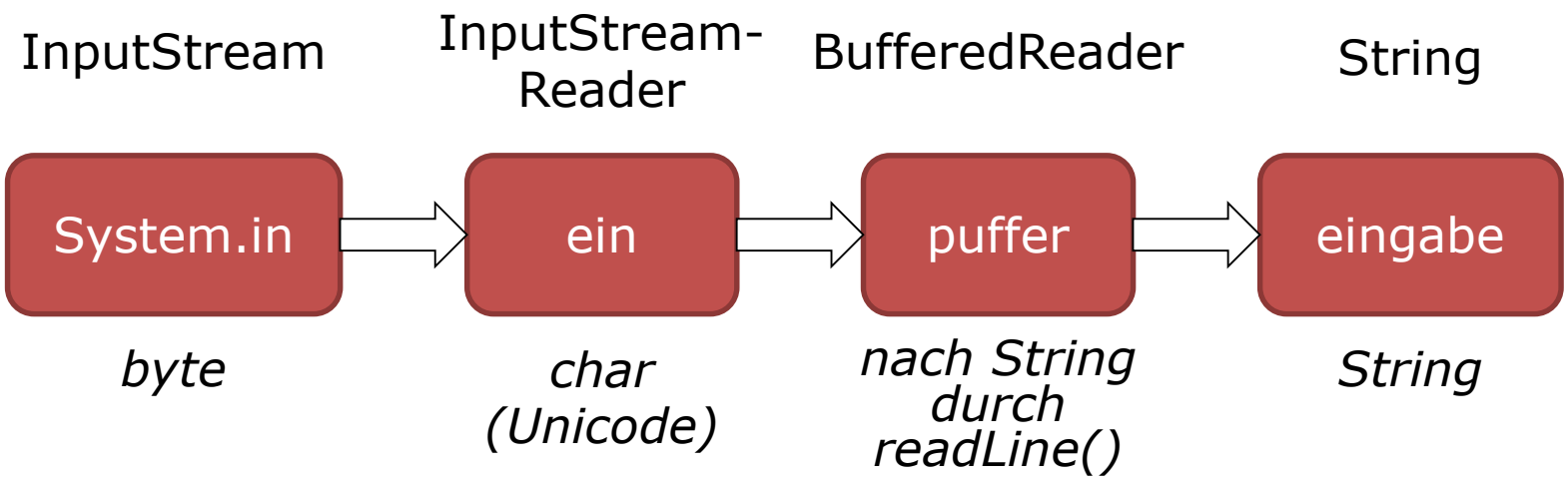
Tastatureingabe (1/3)

```
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());
      }
   }
}
```

→ Mehr als zehn Eingabezeichen werden nicht (zwischen)gespeichert

Tastatureingabe (2/3)

- Bytes des Byte-Array werden in Unicode-Zeichen konvertiert und in ein String-Objekt umgewandelt
- Umweg über ein Byte-Array kann über verknüpfte Eingabeströme vermieden werden



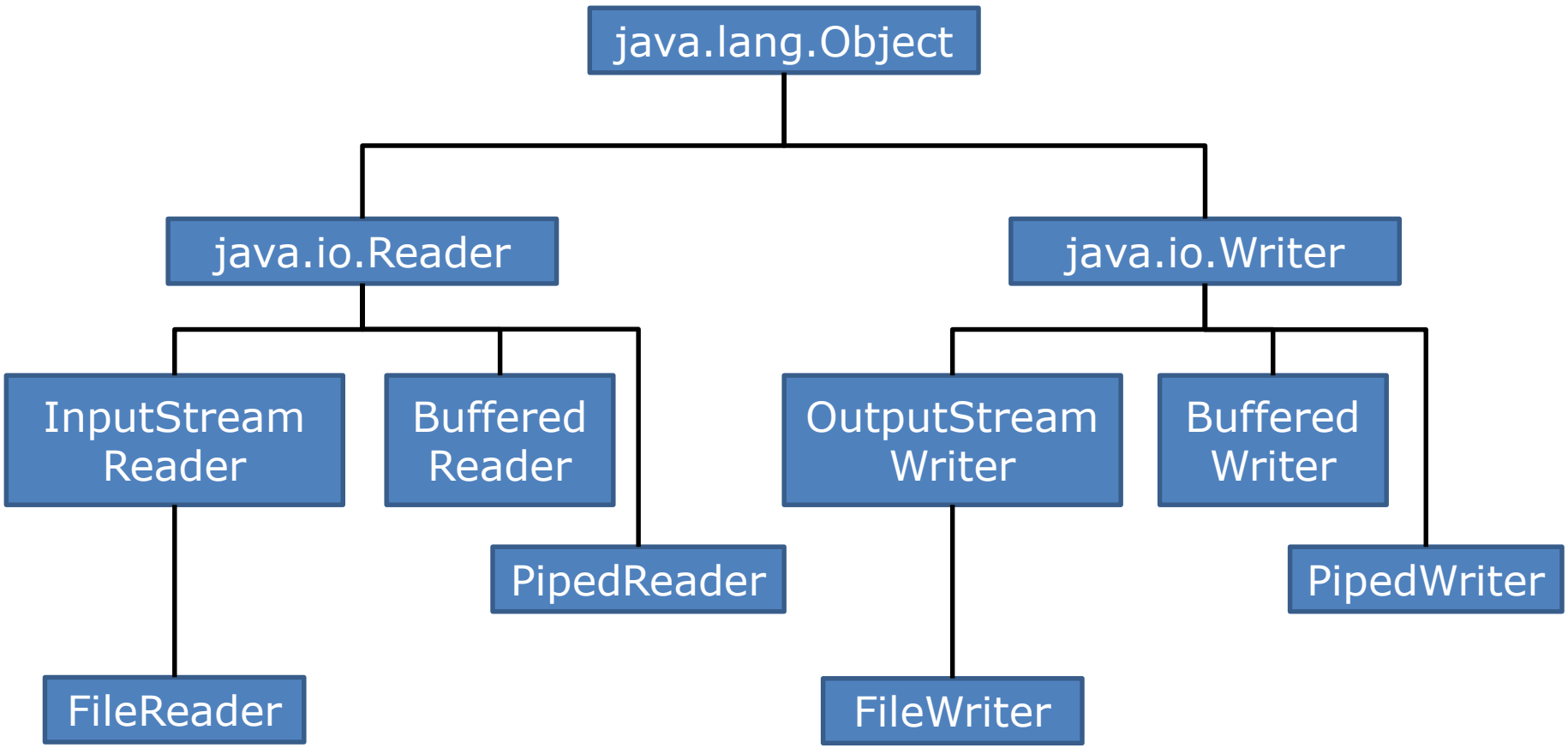
- Anzahl der Zeichen muss nicht zwischen gespeichert werden
- String muss nicht durch Konstruktor erzeugt werden

Tastatureingabe (3/3)

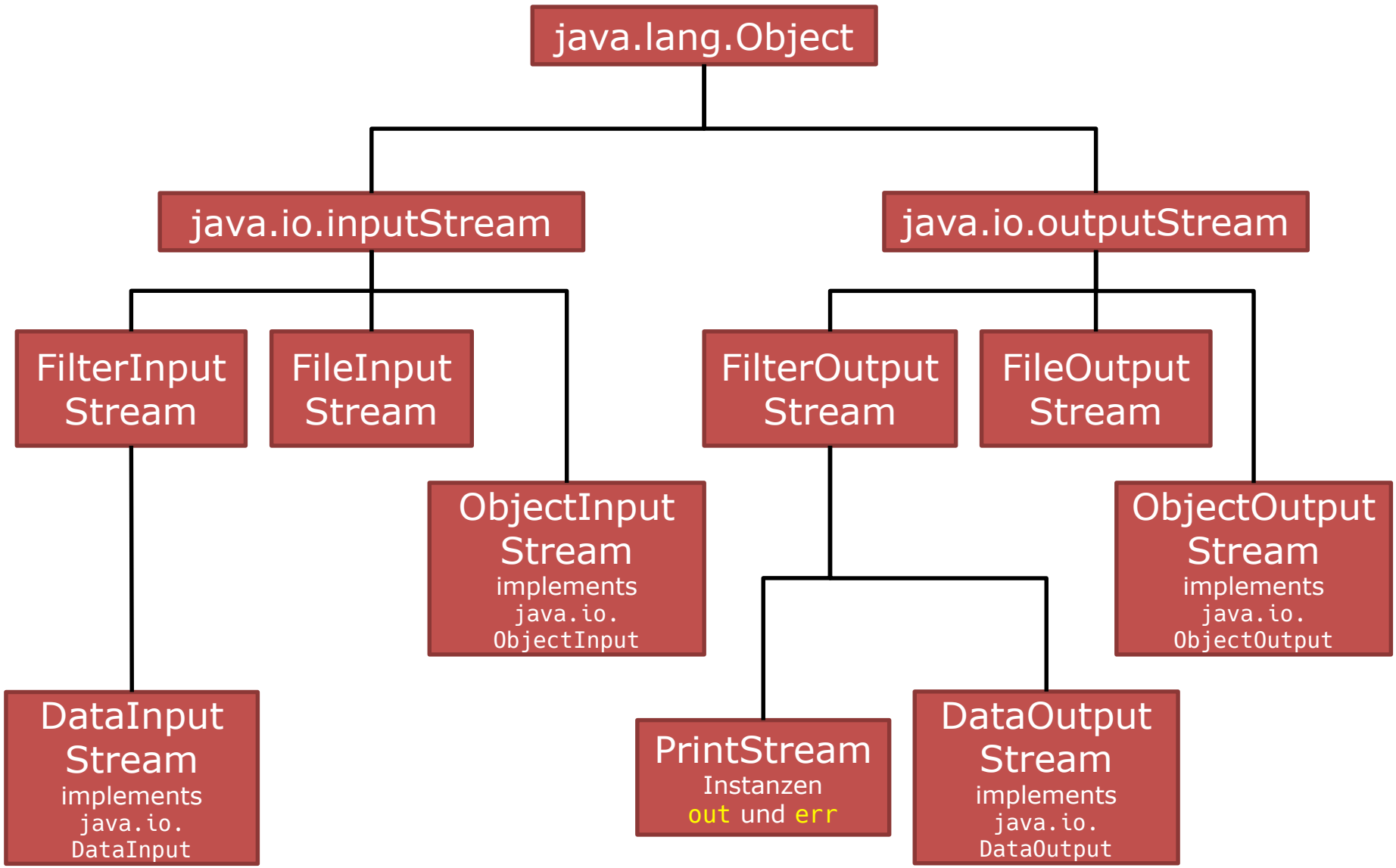
```
/* … */
String Eingabe;
InputStreamReader Ein = new InputStreamReader(System.in);
BufferedReader Tastatur = new BufferedReader(Ein);
System.out.print("Eingabe: ");
try {
   Eingabe = Tastatur.readLine();
   System.out.println(Eingabe);
}
catch (IOException e) {
   System.err.println("Fehler: " + e.getMessage());
}
/* … */
```

- Ausnahmebehandlung wird erzwungen
- Plattformunabhängigkeit begründet Umweg über Byte-Puffer

Hierarchie der IO-Klassen, Unicode-orientiert



Hierarchie der IO-Klassen, Byte-orientiert



Text aus Datei lesen

```
/* ... */
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 in Datei schreiben

```
/* … */
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) {   // String-Vergleich
      Aus.write(Textzeile);
      Aus.newLine();                         // systemabhängiges Zeilenende
      System.out.print("Text eingeben: ");
      Textzeile= Tastatur.readLine();        // Nachlesen Tastatur
   }
   Aus.close();                              // Schließen
}
catch (IOException e) {
   System.err.println("Fehler: " + e.getMessage());
}
/* … */
```

Textdateien

- 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; sonst wird die Datei neu angelegt)
- `BufferedReader` bzw. `BufferedWriter` sorgen dafür, dass nicht jedes Byte einzeln gelesen bzw. geschrieben wird
- `Aus.newLine()` schreibt ein systemabhängiges Zeilenende
- Daten werden erst nach `Aus.close()` auf die Platte geschrieben

Datendateien

- 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 in Dateien zu schreiben bzw. lesen
  - Daten liegen in Datei in durch Programm direkt lesbarem Format
  - solche Dateien sollten nicht außerhalb des Programms geöffnet oder gar verändert werden!
    (wenn, dann mit einem Hex-Editor)

Daten schreiben

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

Daten lesen

```
/* ... */
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

- Lesen des Dateiendes von Datendateien wirft eine Exception
  → muss in einem `catch`-Zweig abgefangen werden
  → `catch`-Zweig kann leer sein
- es gibt keine Methode zum Erstellen von Dateien
  - werden durch die entsprechenden Streams bei Bedarf erzeugt
  - werden durch die entsprechenden Streams bei Bedarf ergänzt
- Paket für einige weit verbreitete Komprimieralgorithmen verfügbar
- es gibt auch Klassen für wahlfreien Zugriff

Objekte serialisieren und laden

- 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 (Umkehrung: Deserialisieren)
  - dazu dienen die Methoden `writeObject` und `readObject`
  - Objekte sind nicht automatisch serialisierbar: Flag-Interface `Serializable` muss implementiert werden
  - nicht zu speichernde Attribute werden mit `transient` gekennzeichnet
  - es wird rekursiv inklusive untergeordnete Objekte serialisiert
    → Jedes Objekt wird aber exakt einmal serialisiert!

Objekte serialisieren

```
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
- Daten sind nicht sinnvoll mit einem Texteditor lesbar

Objekte laden

```
/* … */
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 explizit gecastet werden

Netzwerk

Stufen der Client-Server-Anwendung

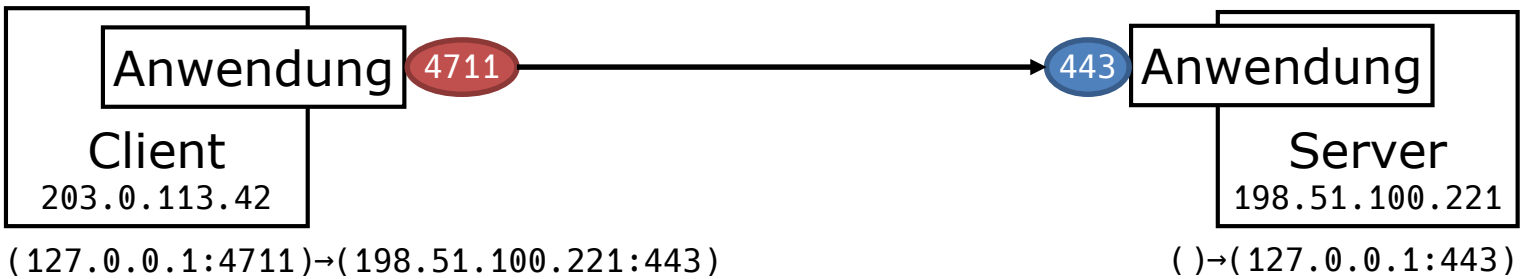
- Übersicht
- Grundgerüst des Servers
  - 1: Listener anlegen
  - 1a: Warten auf Client und Verbindungs-Socket, Ein-/Ausgabe-Objekte anlegen, String-Objekt anlegen
  - 1b: while-Schleife für Verbindung
- Grundgerüst des Client
  - 2: Verbindungs-Socket anlegen
  - 2a: Ein-/Ausgabe-Objekte anlegen, String-Objekte anlegen
  - 2b: while-Schleife für Verbindung

Sockets

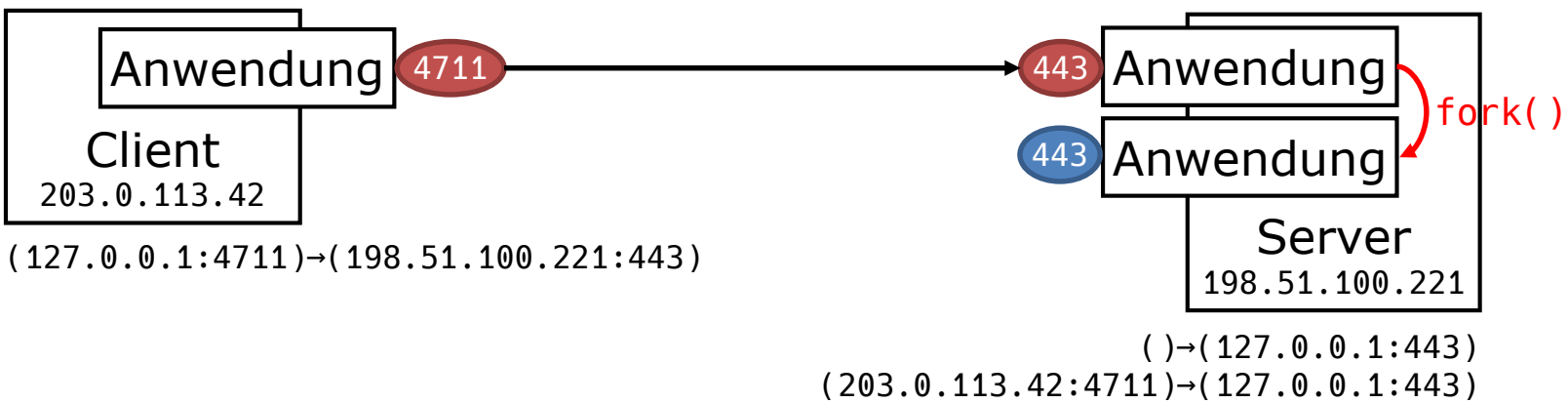
- Rechner werden über IP-Adressen identifiziert
- Kommunikationskanäle repräsentieren Port-zu-Port-Verbindungen
- Endknoten der Kommunikationsverbindungen sind die Sockets (d.h. eindeutige Kombination aus IP-Adresse und Port)
  - es lassen sich 65.535 Ports ansprechen
  - Port-Nummern kleiner 1024 werden historisch als wohl definiert bezeichnet (Beispiele: SSH 22, SMTPs 143, HTTPs 443)
- für Verbindungsaufnahme muss Client *einen* Ziel-Socket kennen
- zuerst werden Server und Client lokal auf einem Rechner ausprobiert (d.h. `127.0.0.1:*`, `[::1]:*` bzw. `localhost:*`)

VERBINDUNGSAUFNAHME

1. Client kontaktiert Server am Listener Socket



2. Server forkt Anwendung und bildet Connected Socket



- i.d.R. hat Connected Socket gleichen Port wie Listener Socket
- Abweichungen sind aber möglich! (Bspw. Active FTP)

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

   }
}
```

1: Listener Socket

1

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

      1a     // VerbindungsSocket, Warten auf Client
             // Ein- Ausgabe-Objekte und Strings

      1b     // while-Schleife zur Verbindung

}
catch (Exception e) {
   System.err.println("Fehler! " + e);
}
```

1A: VERBINDUNGS-SOCKET, E/A-OBJEKTE

1a

```
// Horchen, bis sich Client meldet

System.out.println("Server wartet auf Client");

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

NetzAusgabe = new PrintStream(
          VerbindungsSocket.getOutputStream());

NetzEingabe = new BufferedReader(
          new InputStreamReader(
             VerbindungsSocket.getInputStream()));

String EmpfangZeile = new String();
```

1b: while-Schleife für Verbindung

1b

```
// 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 Clients

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

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

      2  // Programmteil

   }
}
```

2: Verbindungs-Socket

2

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

      2a    // Ein- Ausgabe-Objekte und Strings

      2b    // while-Schleife zur Verbindung

}
catch (Exception e) {
   System.err.println("Fehler! " + e);
}
```

2A: E/A-OBJEKTE, STRINGS

2a

```
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();
```

2b: while-Schleife für Verbindung

2b

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

System.exit(0);
```

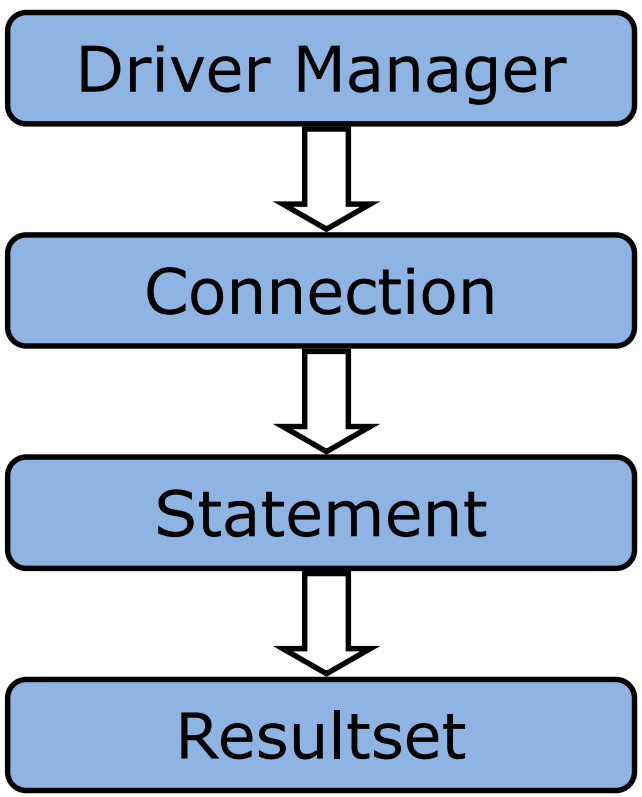
Datenbanken

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

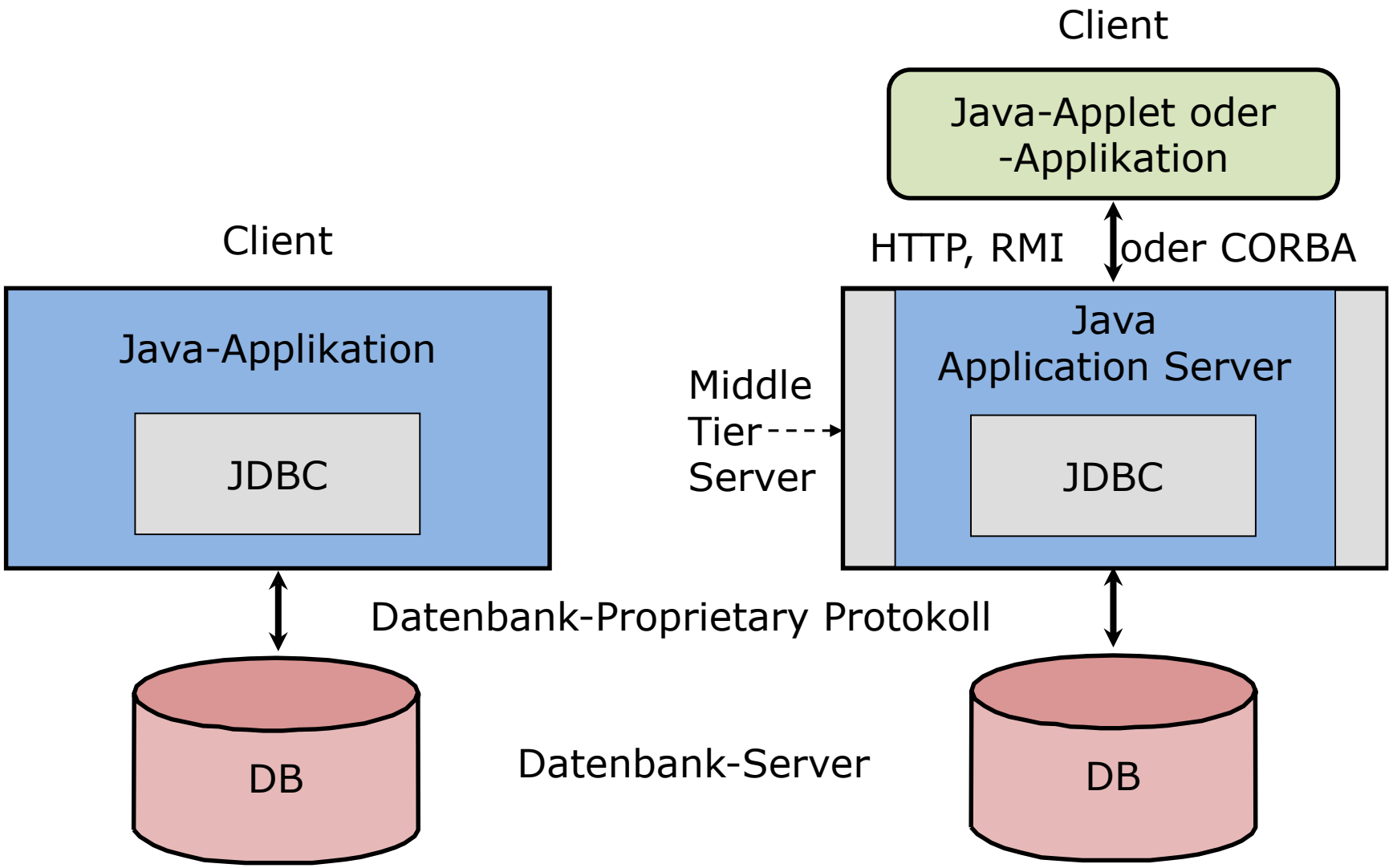
Datenbankanbindung

- JDBC API stehen unter `java.sql` und `javax.sql` zur Verfügung
- 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 Datenbank-abfrage entgegen

Two-Tier- und Three-Tier-Model



JDBC-Treiber (1/3)

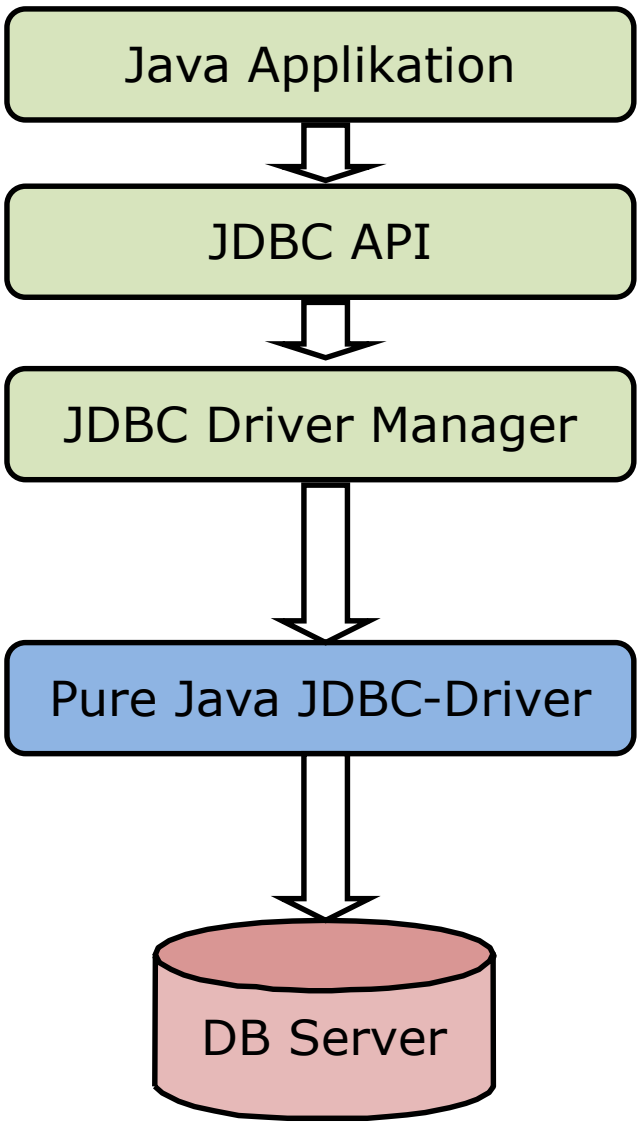
JDBC-Wrapper-Klasse: Unterschiede der Datenbanksysteme werden abgefangen und das Verhalten wird angepasst

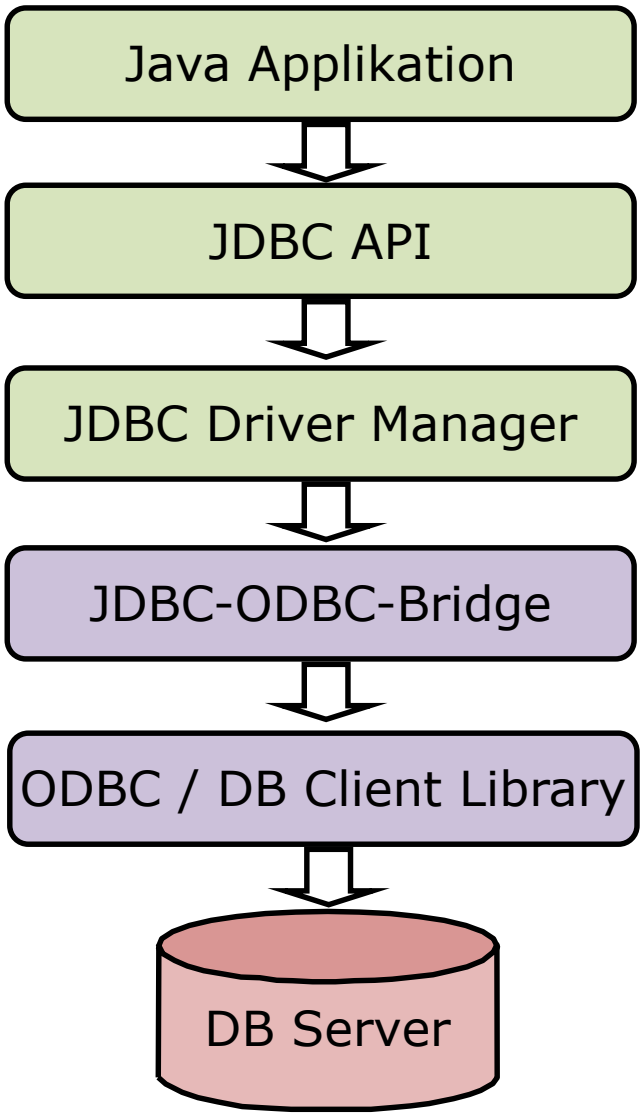
1. JDBC-ODBC Bridge
   - einfacher Zugriff über den Standard-ODBC-Treiber (Open Database Connectivity)
   - muss auf jedem Client-Rechner installiert sein
2. JDBC-Native Treiber
   - „Native-API-partly Java Technology-enabled Driver" greift auf proprietären Treiber des Datenbankherstellers zu
   - muss auf jedem Client-Rechner installiert sein

JDBC-Treiber (2/3)

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)
4. Direktzugriff
   - „Native-protocol fully Java Technology-enabled Driver“
   - JDBC-API-Aufrufe werden direkt in das DBMS-eigene Protokoll des Datenbank-Servers konvertiert

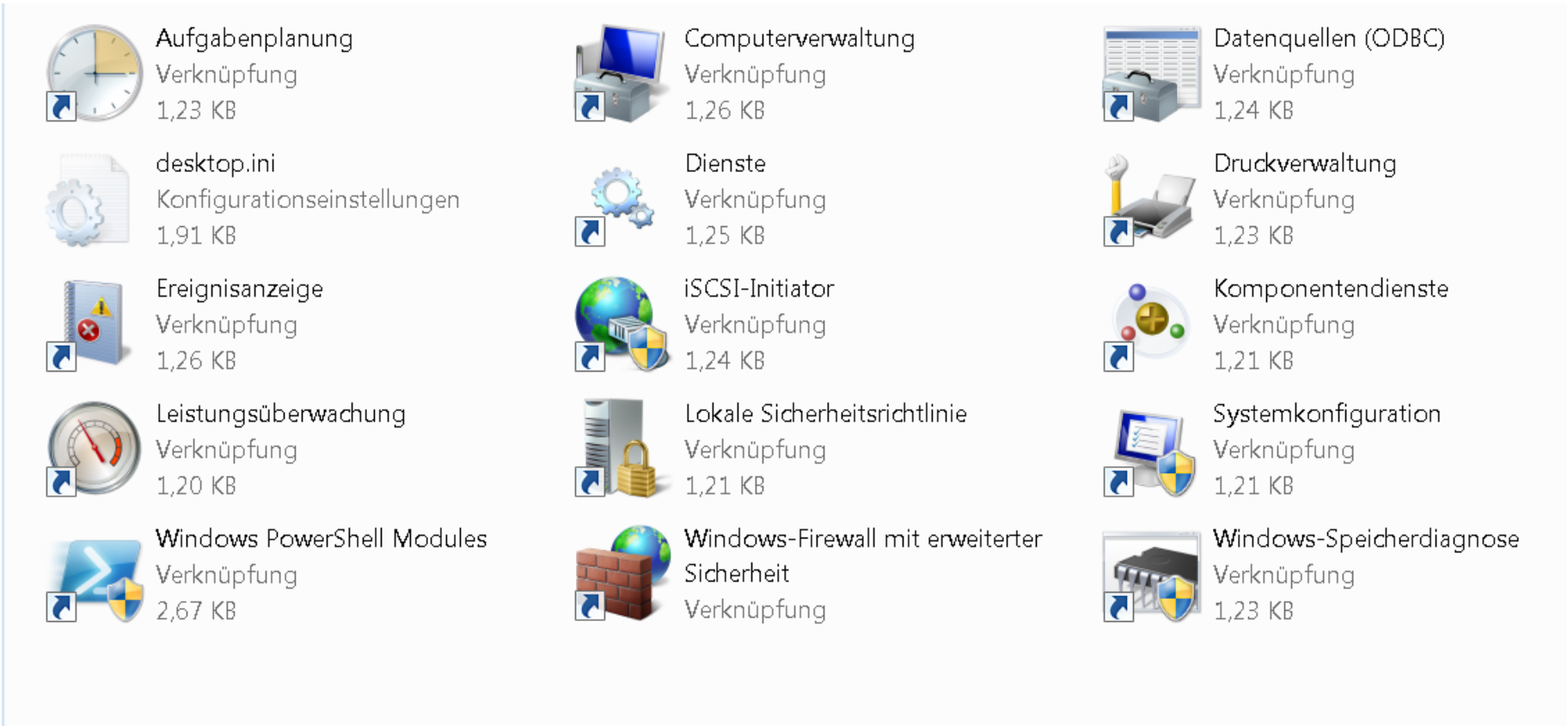
JDBC-Treiber (3/3)



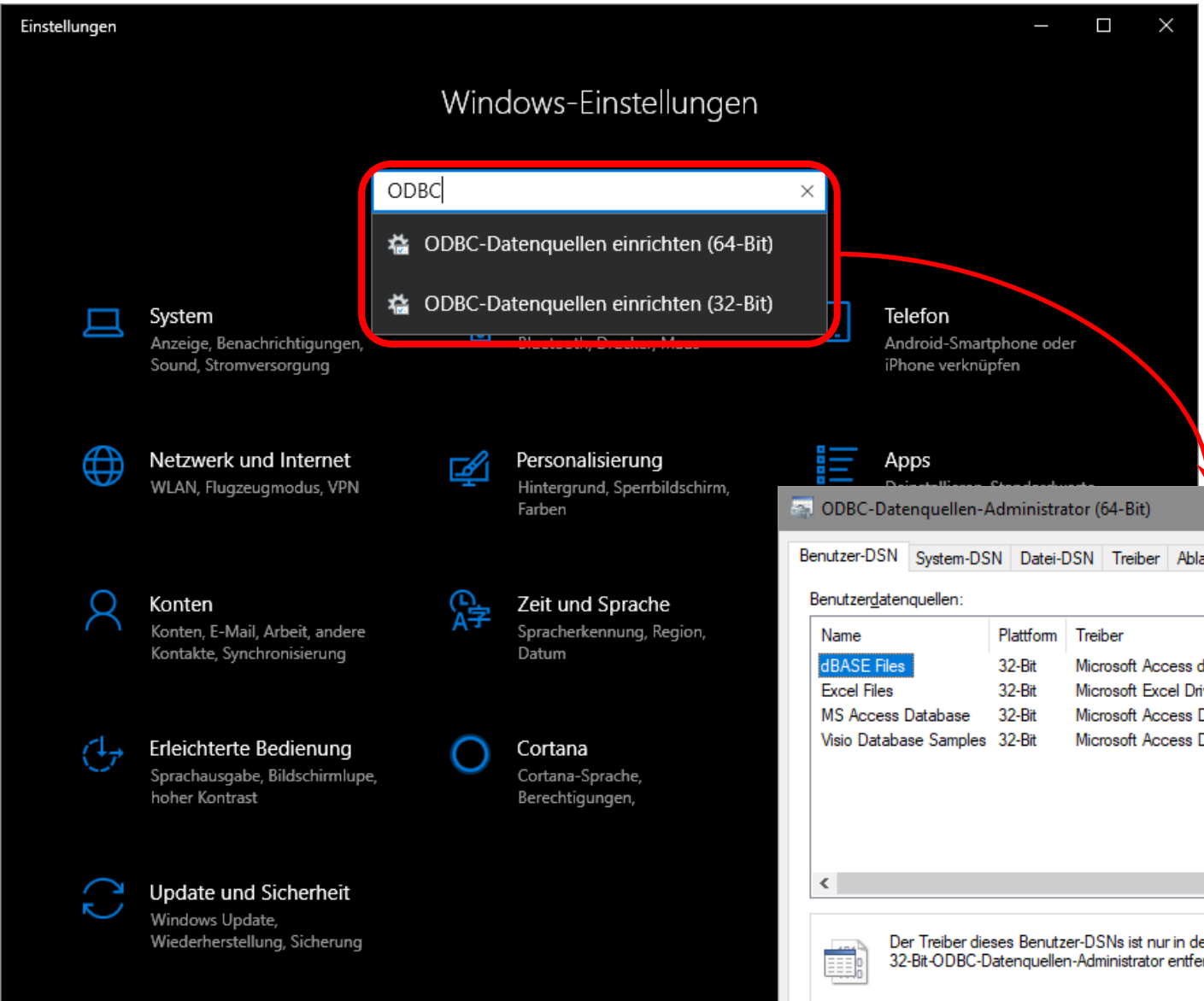


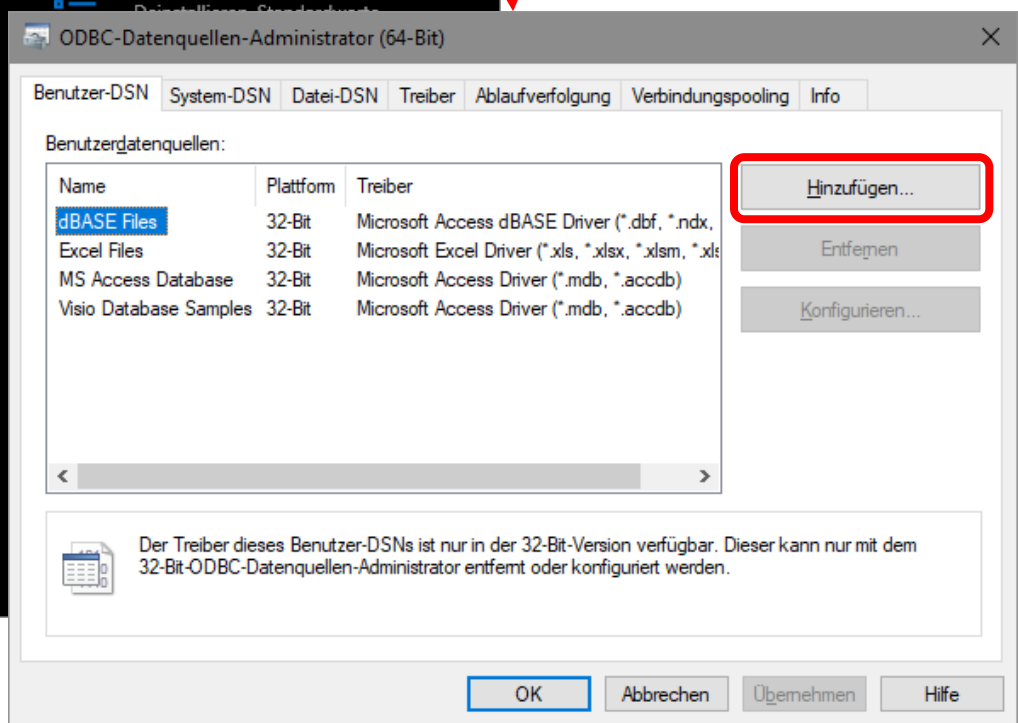
ODBC-Datenbankquelle einrichten

- Windows Systemsteuerung → Verwaltung

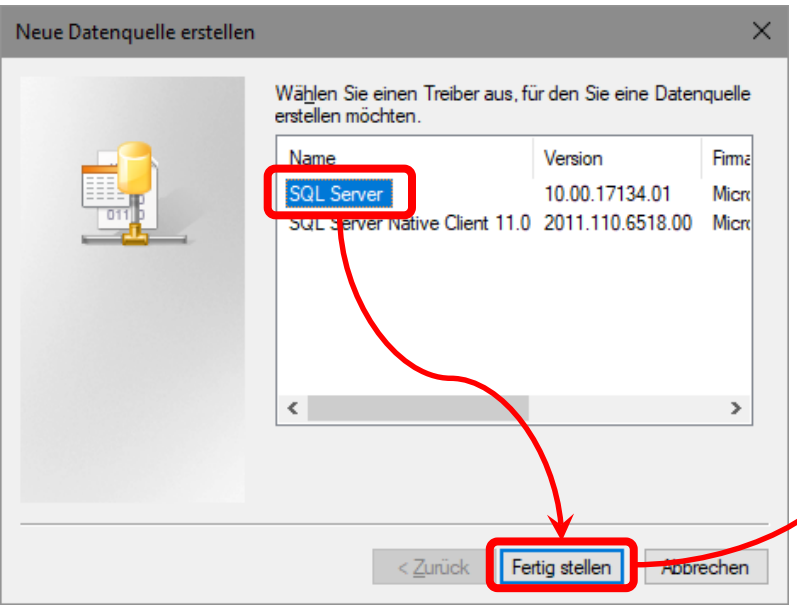


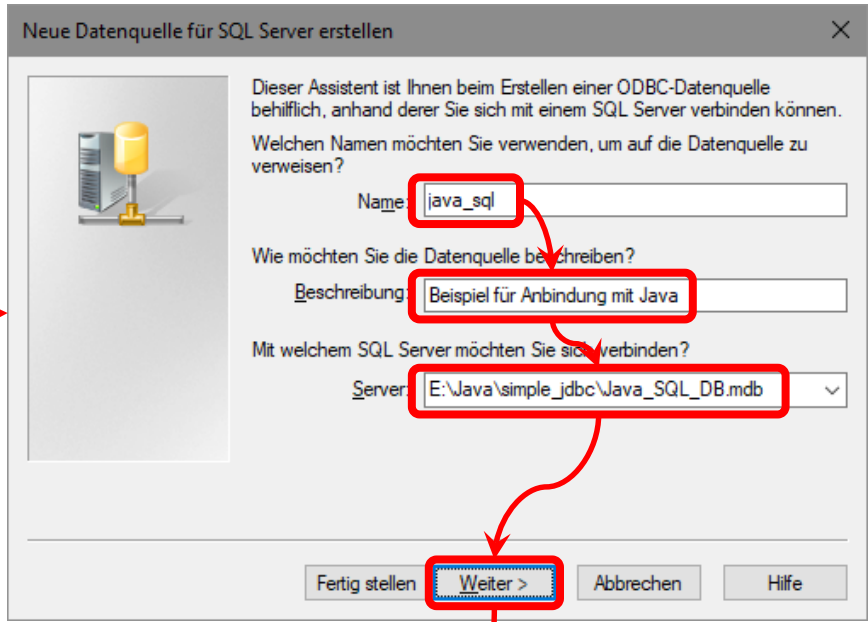
ODBC-Datenbankquelle einrichten (1/2)

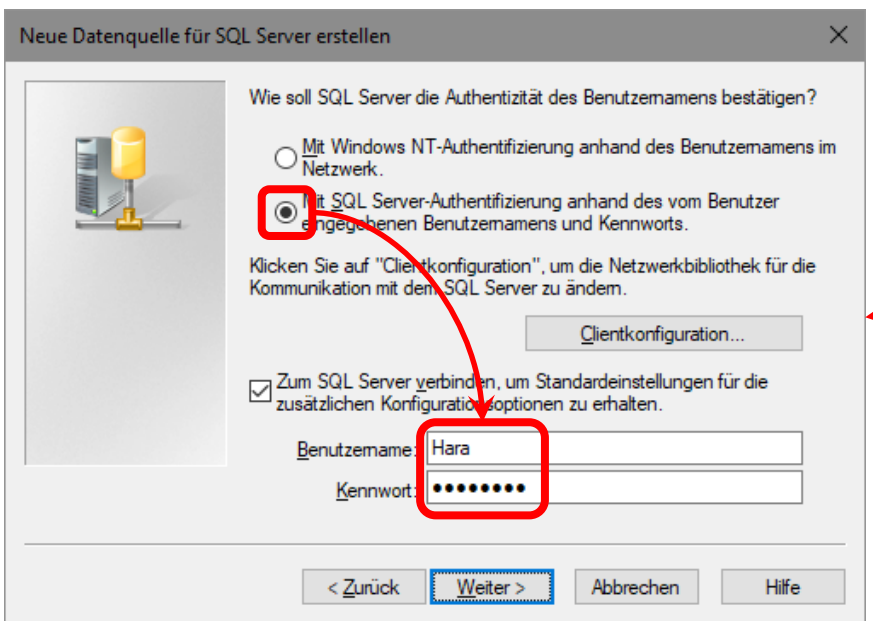




ODBC-Datenbankquelle einrichten (2/2)







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

   }
}
```

1: Datenbank-Treiber laden (2/5)

1a

```
// Laden des ODBC-Treibers muss auch ohne DB fehlerfrei
// funktionieren; ansonsten Fehler einfangen (1b)
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

1b

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

2A, 2C, 3C UND 4B: 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());
}
```

2b, 3 und 4a: Zugriff auf Datenbank (4/5)

```
Statement stmt = db.createStatement();
```
3a

```
ResultSet rs = stmt.executeQuery(
           "SELECT * FROM adressen");
```
4a

5

```
stmt.close();
```
3b

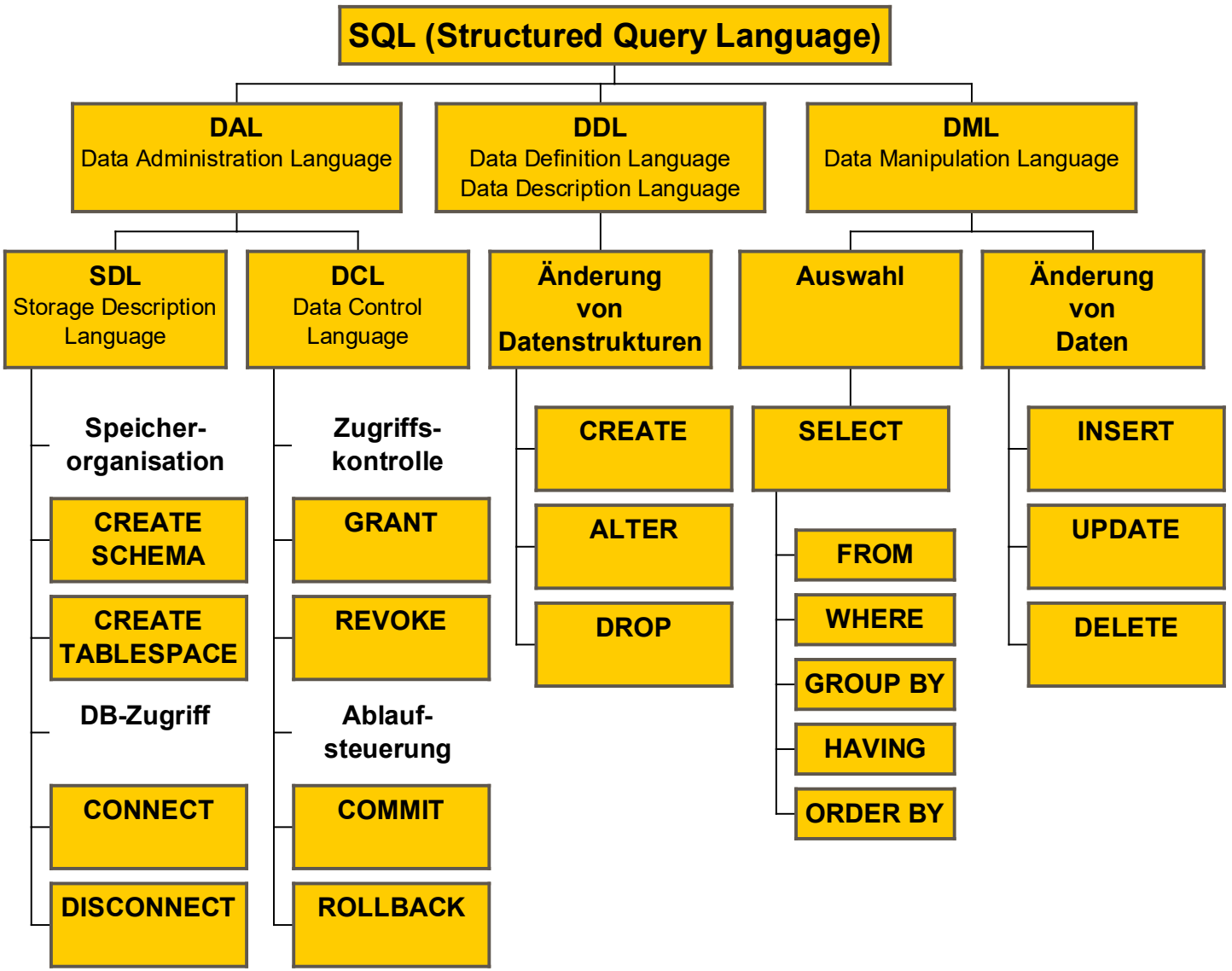
```
db.close();
```
2b

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



Beispielanweisung: SELECT-Abfrage

```
SELECT (Feldname, Feldname, … | * )
FROM Tabelle [, Tabelle, Tabelle, … ]
[ WHERE (Bedingung) ]
[ GROUP BY Feldname [ HAVING (Bedingung) ] ]
[ ORDER BY Feldname [ ASC | DESC ] … ]
```

LITERATURANGABEN

Elke Niedermair / Michael Niedermair: *Internet-Programmierung mit Java*; Data Becker GmbH & Co. KG, ISBN 3-8158-2086-3

Alexander Niemann: *Das Einsteigerseminar*, *Objektorientierte Programmierung in Java*; bhv Verlag Bürohandels- und Verlagsgesellschaft mbH, ISBN 3-8287-1015-8

Java 2 SDK v 1.2.2*, Grundlagen Programmierung*; HERDT-Verlag für Bildungsmedien GmbH, Nackenheim