Hochschule
Bonn-Rhein-Sieg

Datenbanksysteme

Datenbanken in Java – JDBC

Wintersemester 2017 / 2018

Prof. Dr. Sascha Alda
(sascha.alda@h-brs.de)

Motivation

- Bisher: Zugriff auf eine Datenbank nur über Datenbank-Clients (Tools):
  - PhpPGAdmin
  - MySQL-Client
  - Eclipse SQL Explorer
- Wie kann man eine Datenbank aus einem eigenen Programm ansteuern?
- Nahezu jede Programmiersprache bietet spezielle Frameworks für den Zugriff auf Datenbanken an
- Datenbanksysteme sind offen für den Zugriff durch „Fremdprogramme“
  - Meist proprietäre Schnittstellen
  - In alten Datenbanksystemen war dies früher die Regel, es gab keine DBS-eigenen Clients!
- Fokus in Vorlesung: das Framework JDBC zur Programmiersprache Java

Literatur

- Saake, Gunter: „Datenbanken und Java – JDBC, SQLJ, ODMG und JDO“. dpunkt Verlag, 2003

- Online Tutorial mit Beispielen und Treibern (für Übung!)
  - http://jdbc.postgresql.org/

- Christian Ullenboom: „Java ist auch eine Insel.“ Buch oder:
  - http://openbook.galileodesign.de/javainsel5/javainsel20_000.htm

Aufbau der Vorlesung

Kapitel 10: Java und Datenbanken - JDBC	
✓ 1	Motivation
2	**Allgemeiner Überblick über JDBC**
3	Entwicklung mit JDBC
4	Exkurs: Objektrelationales Mapping (ORM)
5	Exkurs: Realisierungsformen einer Datenbankanwendung in Java (N-Tier Architekturen)
6	Zusammenfassung und Ausblick

JDBC – ein Überblick

- Java Database Connectivity (JDBC) ist ein Framework (Sammlung von Klassen) innerhalb der Java-Plattform, das eine einheitliche Schnittstelle zu relationalen Datenbanken verschiedener Hersteller bietet
  - Standardmäßig ab Java 1.1 integriert
  - Package `java.sql.*`
  - Basiert auf einer Spezifikation (aktuell: JDBC 4.0 für Java 6, sonst JDBC 3.0 für Java 5)

- Es sind weitere spezielle Treiber-Klassen notwendig, um den Zugriff auf ein gegebenes Datenbanksystem zu realisieren
  - Implementierung der JDBC-Spezifikation
  - Von den meisten DBS-Herstellern angeboten

- Vorbild: ODBC (Open Database Connectivity)

Herausforderung beim Zugriff auf SQL-Tabellen

- Wiederholung: Eine SQL-Tabellen entspricht einer mathematischen Relation (eine Menge von Tupeln)

- Problem: Java kann zwar einzelne Tupel (Klasse) darstellen, jedoch keine mengenorientierte Relation
  - Nach-Implementierung möglich, jedoch komplex
  - Seit Java 1.2: Interface `java.util.Set` – aber auch hier nur begrenzte Möglichkeiten zur Auswertung einer Menge

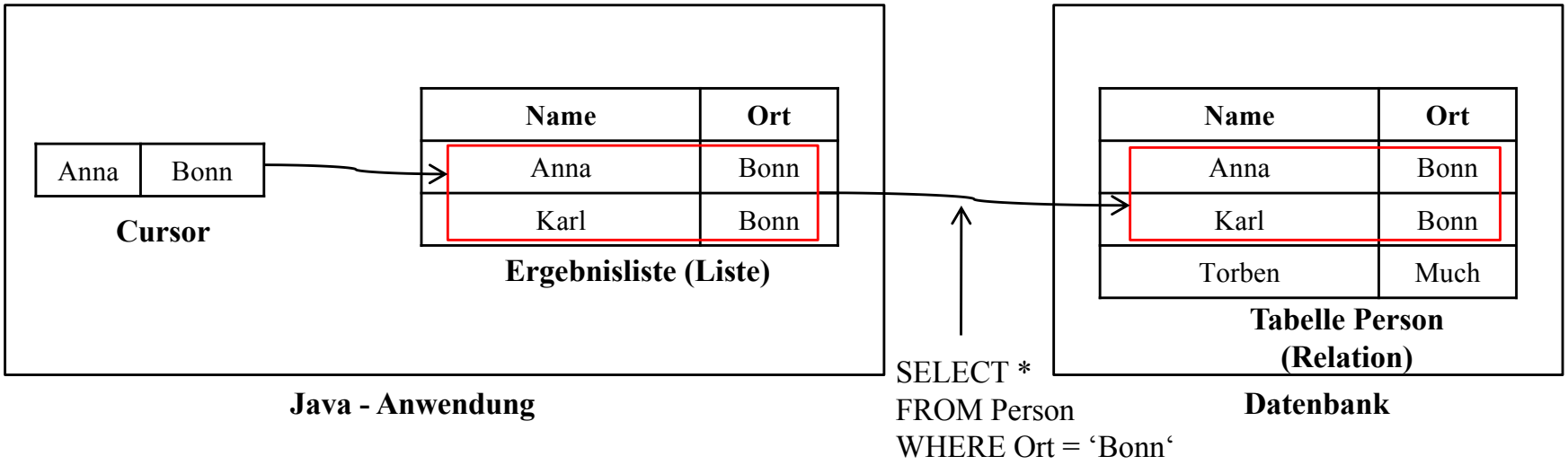
- Ansatz bei JDBC: Relationen werden als abstrakt als Listen realisiert, Zugriff über einen Cursor innerhalb einer Anwendung

- Implementierung des DECLARE CURSOR Anweisung bei SQL.
  - Weitere Infos: http://www.postgresql.org/docs/9.0/static/plpgsql-cursors.html

Ansatz bei JDBC – Darstellung Ergebnislisten

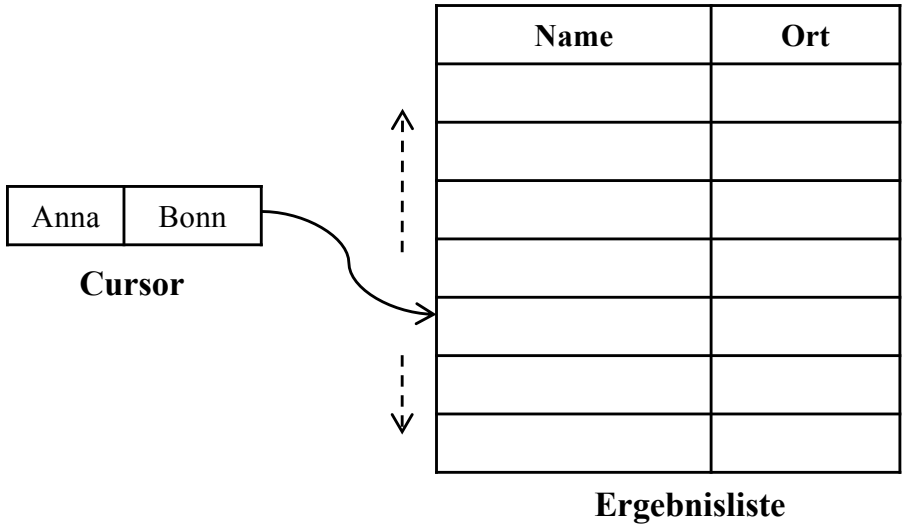
- Ansatz bei JDBC: Relationen werden abstrakt als Listen realisiert, die eine Sicht einer Tabelle aus einer Relationalen Datenbank darstellen

- Navigation über die Liste mit Hilfe eines Cursors



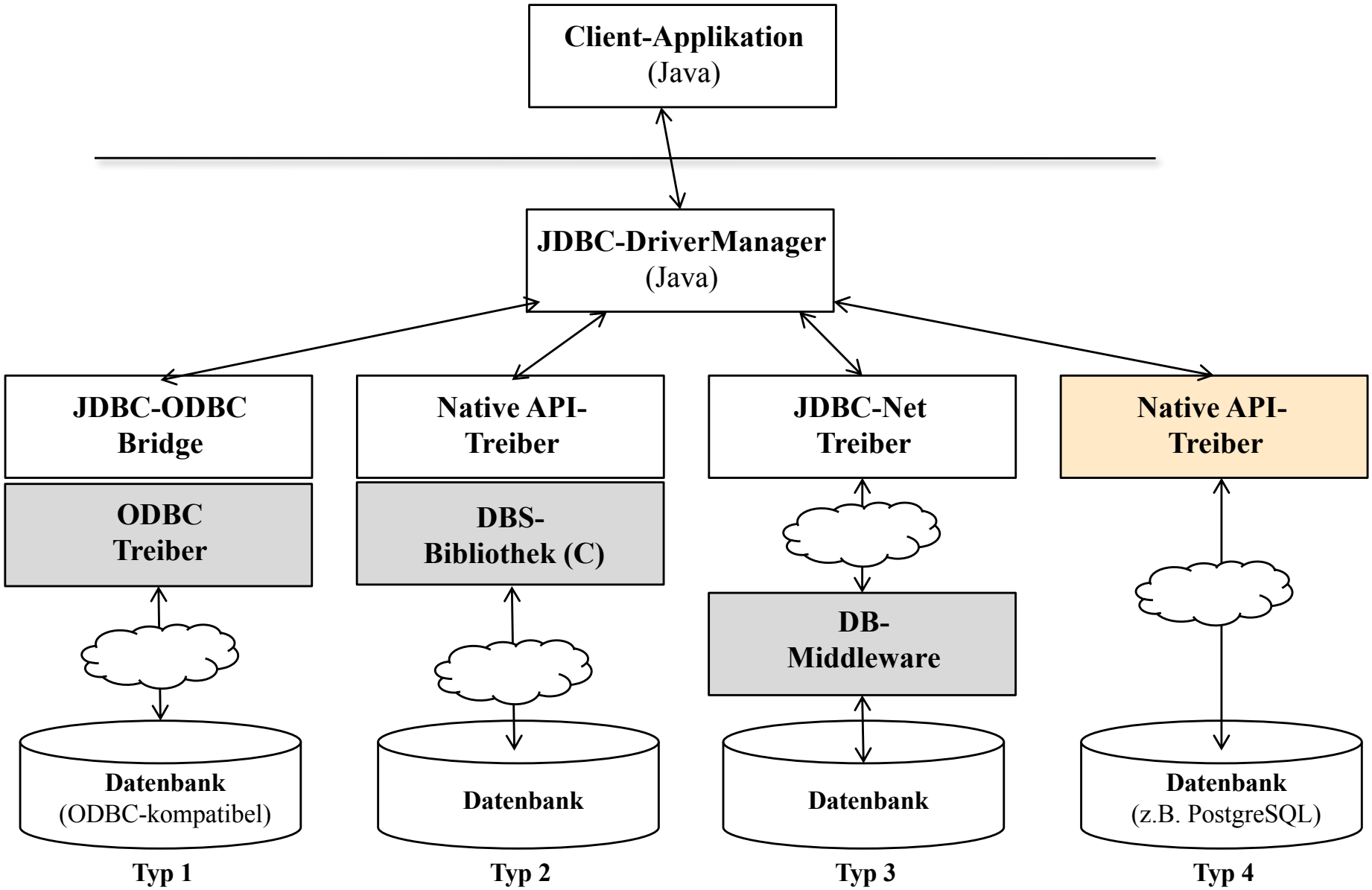
Cursor in JDBC

- Ein Cursor ist ein Iterator über eine Liste von Tupel, d.h. ein Zeiger auf ein Tupel einer Liste, der vor- und zurückgesetzt (seit JDBC 3.0) werden kann



- Über einen Cursor sind alle grundlegenden Operationen möglich:
  - Auslesen eines Tupel
  - Löschen eines Tupel
  - Verändern eines Tupel

Architektur einer JDBC-Applikation mit Treiber-Typen



Aufbau der Vorlesung

Kapitel 10: Java und Datenbanken - JDBC	
✓ 1	Motivation
✓ 2	Allgemeiner Überblick über JDBC
3	**Entwicklung mit JDBC**
4	Exkurs: Objektrelationales Mapping (ORM)
5	Exkurs: Realisierungsformen einer Datenbankanwendung in Java (N-Tier Architekturen)
6	Zusammenfassung und Ausblick

Aufbau einer JDBC-Anwendung

- Eine Verbindung mit dem DBS sowie Abfragen usw. werden aus der Java-Anwendung heraus initialisiert (mit vier Klassen aus `java.sql.*`):

- Die Kommunikation mit dem Datenbanksystem erfolgt durch Treiber-Klassen, die die Schnittstellen der JDBC-Spezifikation implementieren

Klasse (java.sql.*)	Inhalt
DriverManager	Einstiegspunkt, in dem die notwendigen Treiber-Klassen registriert und Verbindungen zur Datenbank aufgebaut werden;
Connection	repräsentiert eine Datenbankverbindung
Statement (PreparedStatement)	ermöglicht die Ausführung von SQL-Anweisungen über eine gegebene Verbindung (Anweisungsobjekt)
ResultSet	verwaltet die Ergebnisse einer Anfrage in Form einer Liste (Ergebnisobjekt)

Aufbau einer JDBC-Verbindung

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

public class SimpleClient {

 public static void main(String[] args) throws SQLException {

  DriverManager.registerDriver( new org.postgresql.Driver() );
  String url = "jdbc:postgresql://dumbo.inf.h-brs.de/demouser“;
  Properties props = new Properties();
  props.setProperty("user",“demouser");
  props.setProperty("password",“demouser");

  Connection conn = DriverManager.getConnection(url, props);

  Statement st;
  st = conn.createStatement();
  ResultSet rs = st.executeQuery("SELECT * FROM bierschema.angebot");

  while (rs.next() ){
    System.out.println( "Kneipe: " + rs.getString( „kneipe“ )); }
  }
}
```

JDBC Treiber registrieren

- Treiber müssen von den Herstellern der Datenbanksysteme (ORACLE, IBM, PostgreSQL) bezogen werden (*.jar* File)
- Setzen des Treibers in der CLASSPATH Umgebungsvariable
- Reinladen des Treibers in Java:

```
DriverManager.registerDriver( new org.postgresql.Driver() );
```

- Alternative: Angabe des Treibers während des Starts des Java-Programms mit dem Parameter –D (CLASSPATH muss auch hier gesetzt sein!)

```
java -Djdbc.drivers = org.postgresql.Driver SimpleClient
```

Verbindungsaufbau

- Übergabe der Netzwerkadresse der Datenbank als URL (Uniform Resource Locator). Syntax:

  `url = jdbc:postgresql:// host : port / database`

- Parameter:
  - *host*: Host-Name des Datenbankservers. Hier: `dumbo.inf.h-brs.de`
  - *port*: Port-Number, unter der der Server „horcht“. Hier: `5432` (optional)
  - *database*: Bezeichung der Datenbank. Hier: `userid`!

- Verbindung:

```
Properties props = new Properties();
  props.setProperty("user",“demouser");
  props.setProperty("password",“demouser");

  Connection conn = DriverManager.getConnection(url, props);
```

Ausführung einer DML Anweisung über JDBC

- Anweisungsobjekt (Statement) erzeugen:

```
Statement st = conn.createStatement();
```

- DML-basierte Abfragen auf eine Datenbank werden mit der Methode `executeQuery( String sql )` ausgeführt

- Konkreter SQL-String mit der genauen Abfragen muss übergeben werden

- Rückgabe in ein `ResultSet`-Objekt:

```
ResultSet rs = st.executeQuery("SELECT * FROM bierschema.angebot");
```

Ausführung einer DDL Anweisung über JDBC

- DDL-Anweisungen zur Erzeugung von Tabellen usw. (CREATE TABLE, ALTER TABLE) werden über die Methode `execute( String sql )` ausgeführt. Beispiel:

```
boolean check = st.execute("CREATE TABLE person ( name
varchar(80) ); ");
```

- DDL-Anweisungen (INSERT, UPDATE, DELETE) werden über die Methode `int executeUpdate( String sql )` ausgeführt. Beispiel:

```
int rows = st.executeUpdate("DELETE FROM angebot WHERE
kneipe='Rheinlust';");
```

- Der Rückgabewert (rows) liefert die Anzahl der betroffenen Zeilen

Weitere Parameter für ein Statement (Auswahl)

- Über die Methode `setQueryTimeout ( int secs )` kann ein Timeout festgelegt werden
  - Default-Wert: 0 (unbegrenztes Warten)
  - Ein Überschreiten eines gesetzten Limits wird durch eine Exception (java.sql.SQLException) signalisiert

- Über die Methode `setMaxRows( int max )` wird die maximale Anzahl von Tupeln im Anfrageergebnis (ResultSet) festgelegt
  - Default-Wert: 0 (keine Einschränkung)

Ermittlung der Ergebnisse aus einem ResultSet

- Ein `ResultSet`-Obekt verwaltet einen internen Cursor, der einen tupelweisen Zugriff auf die Elemente der Menge erlaubt.

- Navigation erfolgt mit der Methode **`boolean next()`**:
  - Liefert so lange **`true`**, wie das Weitersetzen des Cursors erfolgreich war, und damit anzeigt, dass noch weitere Tupel in der Relation vorhanden sind.

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

  // Verarbeitung der einzelnen Tupel
  // Spaltenzugriff...
}
```

Zugriff auf die einzelnen Spalten

- Nach Positionierung des internen Cursors des ResultSet-Objekts kann der Zugriff auf die Spaltenwerte über getXXX Methoden erfolgen
  - (XXX = Datentyp einer Spalte, muss hier bekannt sein)

- Zugriff über Spaltenindex (ab Wert ‚1' aufsteigend!):

```
String kneipe = rs.getString( 1 );
```

- Zugriff über Spaltenname (falls bekannt):

```
String = kneipe rs.getString( „kneipe“ );
```

Typabbildung von SQL nach Java

SQL	get-Methode	Java-Typ
SMALLINT	getShort	short
INTEGER	getInt	int
BIGINT	getLong	long
NUMERIC	getBigDecimal	BigDecimal
DECIMAL	getBigDecimal	BigDecimal
FLOAT	getDouble	double
REAL	getFloat	float
DOUBLE PRECISION	getDouble	double
CHAR	getString	String
VARCHAR	getString	String
BIT	getBoolean	boolean
VARYING BIT	getBytes	byte[]
DATE	getDate	Date
TIME	getTime	Time
TIMESTAMP	getTimestamp	Timestamp

Generische Methode getObject() bei ResultSet

- Mittels der Methode `Object getObject( int collindex )` bzw. `Object getObject( String collname )` wird ein Java-Objekt entsprechend dem SQL-Typ der spezifizierten Spalte zurückgeliefert

- Generischer Zugriff möglich durch Casting:

```
String kneipe = (String) rs.getObject( "kneipe" );
```

Behandlung von NULL-Werten

- Bei normalen Objekten: Direkte Überprüfung auf NULL möglich:

```
Object ob = rs.getObject(“ID");
if ( ob == null ){
  System.out.println("Null-Wert!“);
}
```

- Die Überprüfung, ob ein numerischer Attributwert in der DB auf NULL gesetzt wurde, ist nicht so einfach
  - Abhängig vom DB-Anbieter werden unterschiedliche Werte gesetzt
  - Alternative: Methode `wasNull()`

```
int ID = rs.getInt("ID");
if( rs.wasNull() )
 System.out.println("ID ist auf null gesetzt");
else
 System.out.println("Aktuelle ID: " + ID);
```

Parametrisierte Anweisungen (PreparedStatement)

- PreparedStatement kapselt eine vorkompilierte Anweisung, die bereits während der Erzeugung des Objekts zum Datenbanksystem gesendet wird

- Laufzeitvorteile wenn Anweisungen mehrfach *parametrisiert* ausgeführt werden müssen (z.B. mehrfache INSERTs oder Anfragen)

```
private void queryPrepared() throws SQLException {
  ...
  Connection conn = DriverManager.getConnection(...);

  PreparedStatement stmt = conn.prepareStatement("INSERT
         INTO studenten VALUES (?, ?, ?, ?)");

  stmt.setString(1, "Karl");
  stmt.setString(2, “Müller");
  stmt.setString(3, "Bonn");
  stmt.setInt(4, 21);

  stmt.executeUpdate();
  // kann beliebig oft wiederholt werden
}
```

Parametrisierte Anweisungen (PreparedStatement)

- NULL-Werte werden über die Methode `setNull( int paramIdx, int nullType )` übermittelt. Angabe der Typkonstante aus `java.sql.Types`

```
private void queryPrepared() throws SQLException {
  ...
  Connection conn = DriverManager.getConnection(...);

  PreparedStatement stmt = conn.prepareStatement("INSERT
         INTO studenten VALUES (?, ?, ?, ?)");

  stmt.setString(1, "Karl");
  stmt.setString(2, “Müller");
  stmt.setNull(3, java.sql.Types.NULL);
  stmt.setInt(4, 21);

  stmt.executeUpdate();
  // kann beliebig oft wiederholt werden
}
```

Parametrisierte Anweisungen (PreparedStatement)

- Ausführung von parametrisierten Abfragen mit der Methode `executeQuery()`. Parametrisierung nur von Attributswerten möglich:

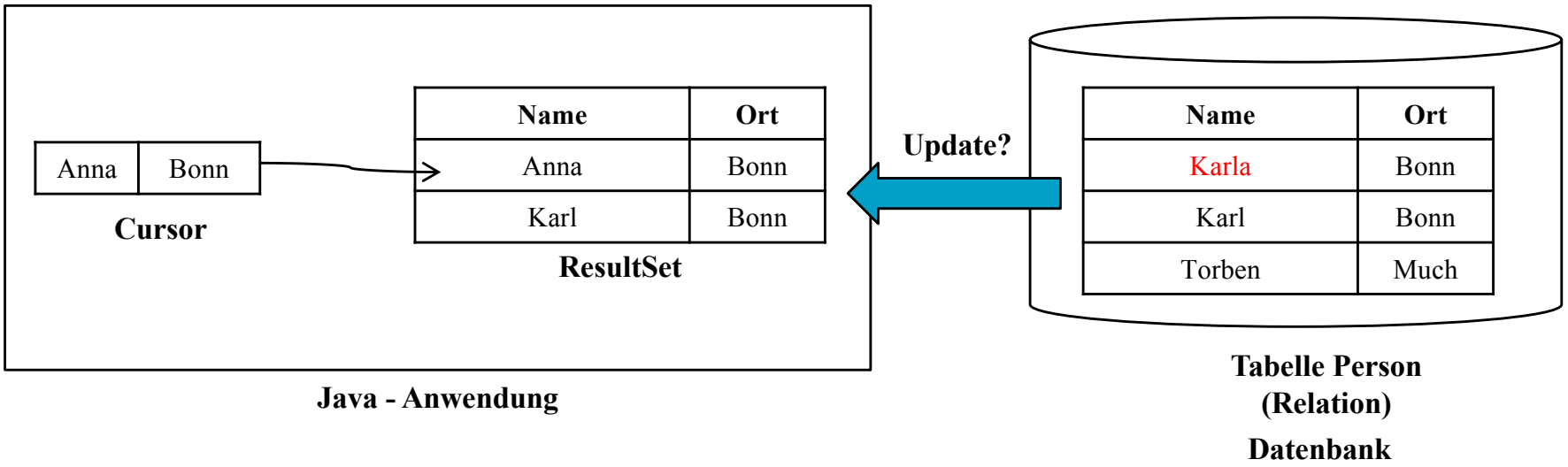
```
...
PreparedStatement stmt2 =
       conn.prepareStatement("SELECT * from studenten WHERE name = ? ");

stmt2.setString(1, “Maier");
ResultSet rs = stmt2.executeQuery();

while (rs.next() ){
   System.out.println( "Name: " + rs.getString( “name" ));
}
...

}
```

Typabbildung von Java nach SQL

Java-Typ	set-Methode	SQL
byte	setByte	SMALLINT
short	setShort	SMALLINT
int	setInt	INTEGER
long	setLong	BIGINT
BigDecimal	setBigDecimal	NUMERIC
float	setFloat	REAL
double	setDouble	DOUBLE
String	setString	VARCHAR
boolean	setBoolean	BIT
byte[]	setBytes	VARYING BIT
Date	setDate	DATE
Time	setTime	TIME
Timestamp	setTimestamp	TIMESTAMP

Erweiterte ResultSets – ScrollType (besser: Änderungstyp)

- `ResultSets` können weiter parametrisiert werden, um das Änderungs-verhalten der Liste festzulegen, falls es auf Seiten der Datenbank zu Änderungen kommt



- Definition über den Scroll-Type Parameter
  - Festlegung zudem, ob man in einer Liste vorwärts und rückwärts navigieren (scrollen) kann

Erweiterte ResultSets - ScrollType

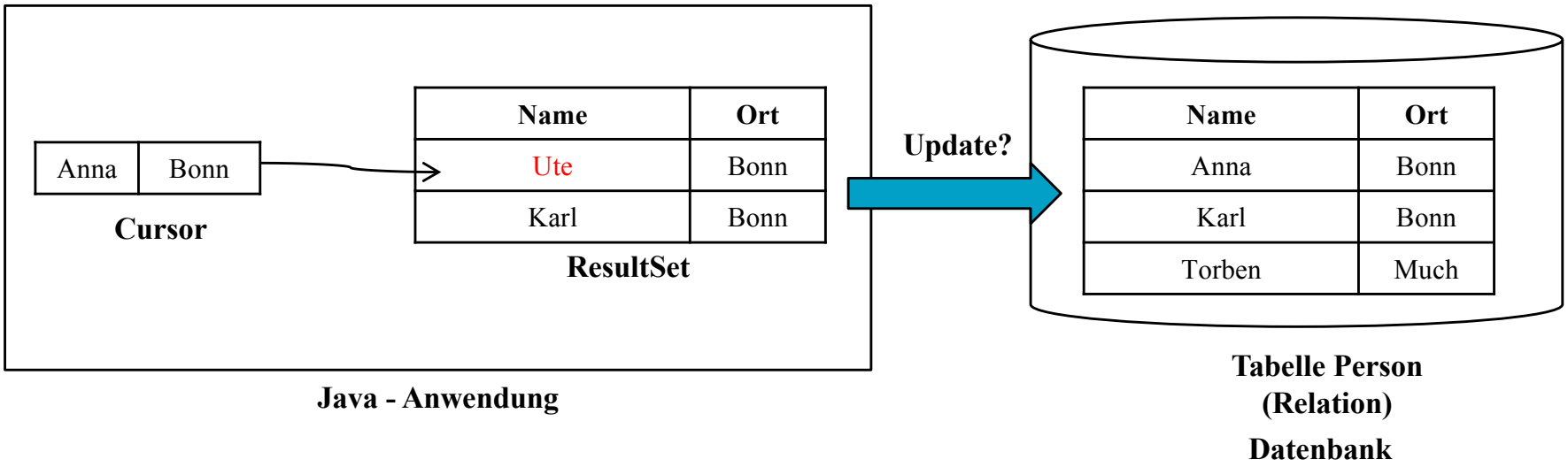
- ScrollTypen (statische Klassen-Variablen aus der Klasse `ResultSet`)
  - TYPE_SCROLL_INSENSITIVE
    - ResultSet wird nach Änderung des Datenbestandes *nicht* geändert
    - Statische Sicht (`ResultList` ist reine Kopie)
    - vorwärts- und rückwärts scrollbar (navigierbar)
  - TYPE_SCROLL_SENSITIVE
    - ResultSet wird nach der Änderung des Datenbestandes geändert
    - Dynamische Sicht (`ResultList` ist mit Datenbank synchronisiert)
    - vorwärts- und rückwärts scrollbar
  - TYPE_FORWARD_ONLY (Default)
    - ResultSet wird nach Änderung des Datenbestandes nicht geändert
    - Statische Sicht
    - nur vorwärts scrollbar

Wie werden Updates aus einer DB in ein ResultSet gebracht?

- Übernahme der neuen Tupelwerte beim Scrollen (z.B. mit `next()`)
- Explizite Angabe eines Refreshs (Methode `refreshRow()`)
- Überprüfung, ob sich ein Tupel an der aktuellen Cursorposition nach der initialen Erstellung bzw. nach dem letzten Update geändert hat:
  - `boolean rowDeleted()`
  - `boolean rowInserted()`
  - `boolean rowUpdated()`
- Aus-Implementierung und somit Funktionalität der Methoden oft unterschiedlich und abhängig vom Datenbanksystem

Erweiterte ResultSets - Synchronisation

- `ResultSets` können weiter parametrisiert werden, um festzulegen, ob ein `ResultSet` geändert werden darf und ob diese Änderungen an eine Datenbank weitergeleitet werden dürfen (Synchronisationsverhalten)



- Definition durch Synchronisations-Typen

Erweiterte ResultSets - Synchronisation

- Synchronisations-Typen (statische Klassen-Variablen aus der Klasse `ResultSet`):
  - CONCUR_READ_ONLY
    - Keine Änderungen auf dem `ResultSet` sind zulässig
  - CONCUR_UPDATABLE
    - Änderungen der Daten sowie das Löschen und Einfügen von Löschen von Tupeln sind möglich
    - Weiterleitung an die Datenbank möglich

Implementierung eines ResultSets - Beispiele

- Implementierung eines `ResultSet`-Objekts, das änderbar und insensitiv ist:

```
Statement st;
st = conn.createStatement( ResultSet.TYPE_SCROLL_INSENSITIVE,
                           ResultSet.CONCUR_UPDATABLE );

ResultSet rs = st.executeQuery(„SELECT * FROM angebot ) ;
```

- Implementierung eines `ResultSet`-Objekts, das nicht änderbar und nur vorwärtsnavigierbar ist:

```
Statement st;
st = conn.createStatement( ResultSet.TYPE_FORWARD_ONLY,
                           ResultSet.CONCUR_READ_ONLY );
// oder:
st = conn.createStatement();

ResultSet rs = st.executeQuery(„SELECT * FROM angebot ) ;
```

Methoden zur Navigation innerhalb eines ResultSet

Methoden (aus ResultSet)	Cursor-Position
`beforeFirst()`	Vor dem ersten Tupel
`first()`	Erstes Tupel
`afterLast()`	Nach dem letzten Tupel
`last()`	Letztes Tupel
`next()`	Nächstes Tupel
`previous()`	Vorheriges Tupel
`absolute( int zeile )`	Tupels auf Indexposition p
`relative( int zeile )`	Tupels auf Indexposition (aktuelle Position + p)

- Bestimmung der aktuellen Position des Cursors:
  - `int getRow()`
  - `boolean isLast(), boolean isFirst(), boolean isAfterLast(), boolean isBeforeFirst()`

Aktualisierung von Tupel in einem ResultSet

- Ähnlich wie bei einem VIEW in SQL sollten `ResultSets` nur dann geändert werden, falls folgende Eigenschaften vorhanden:
  - Die Anfrage beinhaltet nur eine Relation und kein JOIN-Operationen
  - Alle Attributswerte zum Primärschlüssel werden in das Ergebnis übernommen (zur eindeutigen Identifikation)

- Änderungen werden auf das Tupel an der aktuellen Cursorposition mit der Methode `updateXXX()` durchgeführt (XXX = Typ der Spalte)
- Definitive Änderung in DB erst durch Methode `updateRow()`:

```
ResultSet rs = st.executeQuery("SELECT * from \"Department\"");

rs.first();
rs.updateString("Name", “FB Informatik");
rs.updateRow();

System.out.println(“Tupel aktualisiert");
```

Hinzufügen von Tupel in einem ResultSet

- Das Einfügen von Tupeln erfolgt an einer speziellen Einfügeposition, die mit `moveToInsertRow()` erreicht wird.
- Das „Einfügetupel“ wird über die `updateXXX()` Methoden eingefügt.
- Definitive Änderung in DB erst durch Methode `insertRow()`:

```
ResultSet rs = st.executeQuery("SELECT * from \"Department\"");

rs.first();
rs.moveToInsertRow();
rs.updateString("Name", “Fachbereich Journalismus");
rs.updateInt( "ID", 5 );
rs.insertRow();

System.out.println(“Tupel eingefügt");
```

Löschen von Tupel in einem ResultSet

- Für das Löschen von Tupel muss der Cursor entsprechend positioniert werden.

- Löschen aus ResultSet sowie definitives Löschen aus DB erfolgt durch die Methode `deleteRow()`:

```
ResultSet rs = st.executeQuery("SELECT * from \"Department\"");

// Löschen des Eintrags des FB 5
while (rs.next() ){
   int idInt = rs.getInt("ID");
   if (idInt == 5) {
       rs.deleteRow();
   }
}

System.out.println("Tupel Gelöscht");
```

Schließen einer Verbindung

- Nachdem eine Anweisung ausgeführt wurde und die Ergebnisse ermittelt wurden, sollten damit verbundene Ressourcen (Client wie DB-seitig) durch Aufruf der close-Methode freigegeben werden:
  - `ResultSet`
  - `Statement`
  - `Connection`
- Falls nicht: Verbindung wird automatisch durch Garbage Collector geschlossen (evt. verzögert)

```
public class SimpleClient {

 public static void main(String[] args) throws SQLException {
   ....

   rs.close()
   st.close();
   conn.close();
 }
```

Metadaten über ein ResultSet

- Ein Objekt der Klasse `ResultSetMetaData` liefert übergreifende Informationen über ein `ResultSet`
- Abrufbar über ein ResultSet über die Methode `getMetaData()`

- Fokus auf Eigenschaften von Spalten:
  - `int getColumnCount();`
  - `String getColumnName( int col );`
  - `String getColumnTypeName( int col );`
  - u.a.

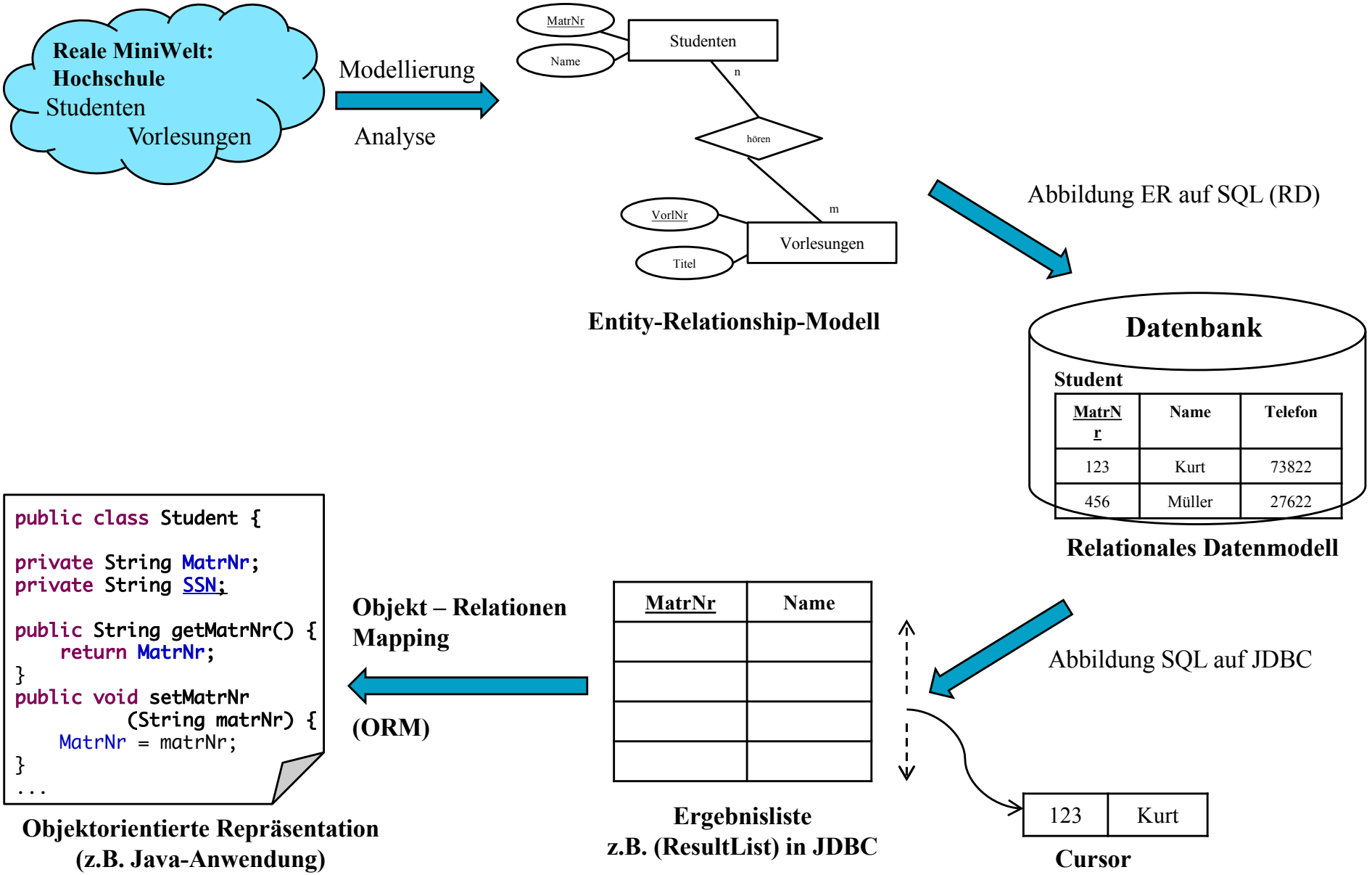
- Weitere Infos:
  - Tabellenname, Schemaname (nicht immer unterstützt)

- Was fehlt: die Anzahl der Tupel im `ResultSet`
  - Kann man trotzdem über die Cursor-Technik rausfinden (→ Übung)

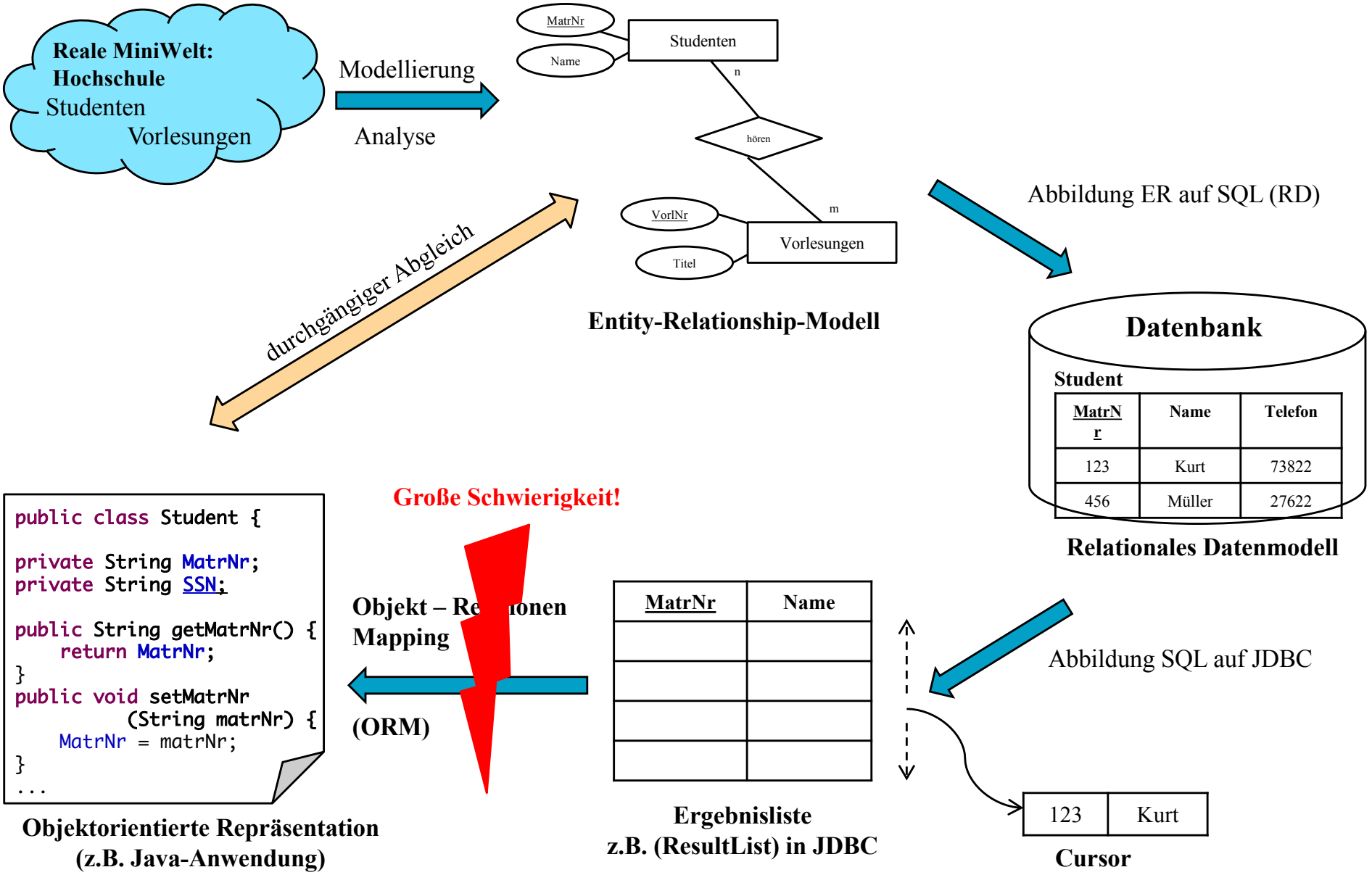
Aufbau der Vorlesung

Kapitel 10: Java und Datenbanken - JDBC	
✓ 1	Motivation
✓ 2	Allgemeiner Überblick über JDBC
✓ 3	Entwicklung mit JDBC
4	**Exkurs: Objektrelationales Mapping (ORM)**
5	Exkurs: Realisierungsformen einer Datenbankanwendung in Java (N-Tier Architekturen)
6	Zusammenfassung und Ausblick

Objekt-Relation Mapping – ein Überblick



Objekt-Relation Mapping – ein Überblick




```
public class Student {

private String MatrNr;
private String SSN;

public String getMatrNr() {
    return MatrNr;
}
public void setMatrNr
          (String matrNr) {
    MatrNr = matrNr;
}
...
```

Objekt-Relation Mapping (ORM)

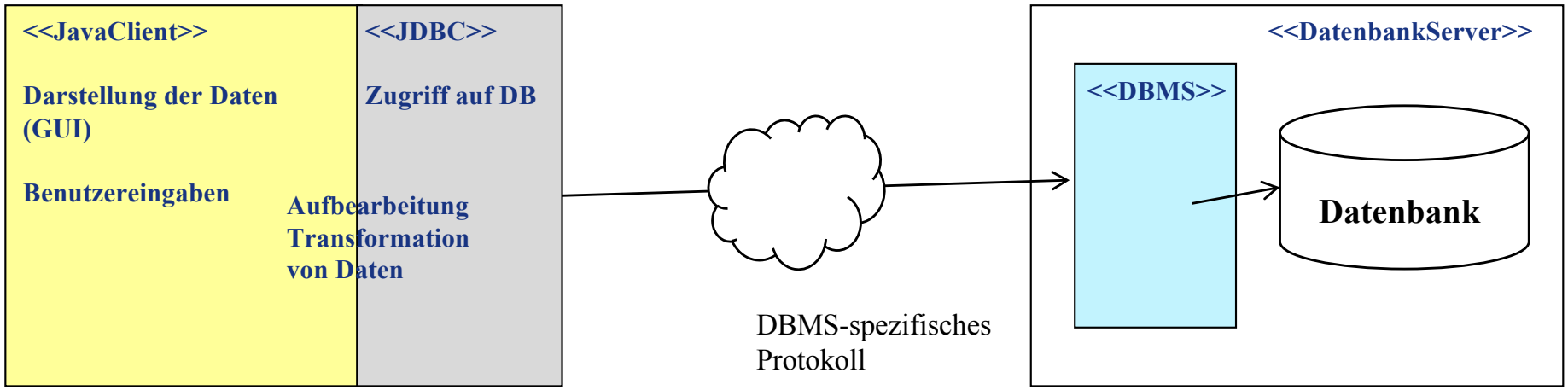
- Problem:
  - Defacto-Standard für Datenmodell: Relationales Datenmodell
  - Defacto-Standard für Programmierung: Objektorientiertes Modell
- Beide Welten sind nicht kompatibel (Object-Relational Impedence Mismatch)
- Die Abbildung (Mapping) zwischen diesen beiden Welten ist (immer noch) eine große Herausforderung
- Technologien zur Abbildung einer relationalen Datenstruktur auf eine objektorientierte Struktur nennt man ORM-Technologien
- State-Of-The-Art Technologien (Auswahl):
  - Hibernate (http://www.hibernate.org/)
  - Java Persistence API (JPA) (http://www.oracle.com/technetwork/articles/javaee/jpa-137156.html)

Aufbau der Vorlesung

Kapitel 10: Java und Datenbanken - JDBC	
✓ 1	Motivation
✓ 2	Allgemeiner Überblick über JDBC
✓ 3	Entwicklung mit JDBC
✓ 4	Exkurs: Objektrelationales Mapping (ORM)
5	**Exkurs: Realisierungsformen einer Datenbankanwendung in Java (N-Tier Architekturen)**
6	Zusammenfassung und Ausblick

2-Tier Architektur

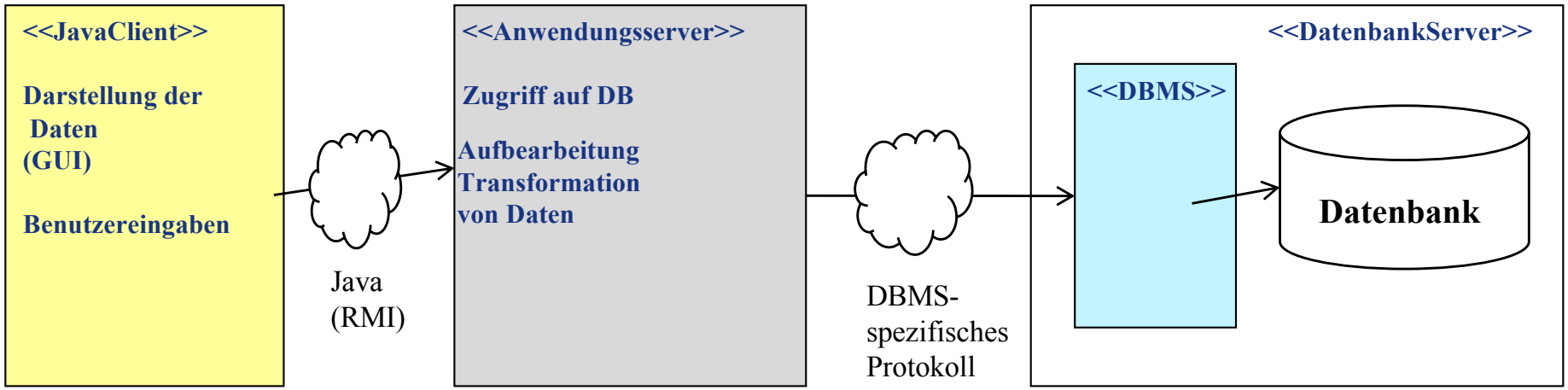
- In einer 2-Tier Architektur befindet sich die komplette Logik für den Zugriff, Darstellung, Transformation von Daten auf der Client-Seite (Fat-Client)



- Vorteile:
  - Gute Architektur für Anwendungsszenarien mit hohem Datenvolumen
- Nachteile:
  - Bei vielen Clients kann es zu komplexen Anpassungen kommen

3-Tier Architektur

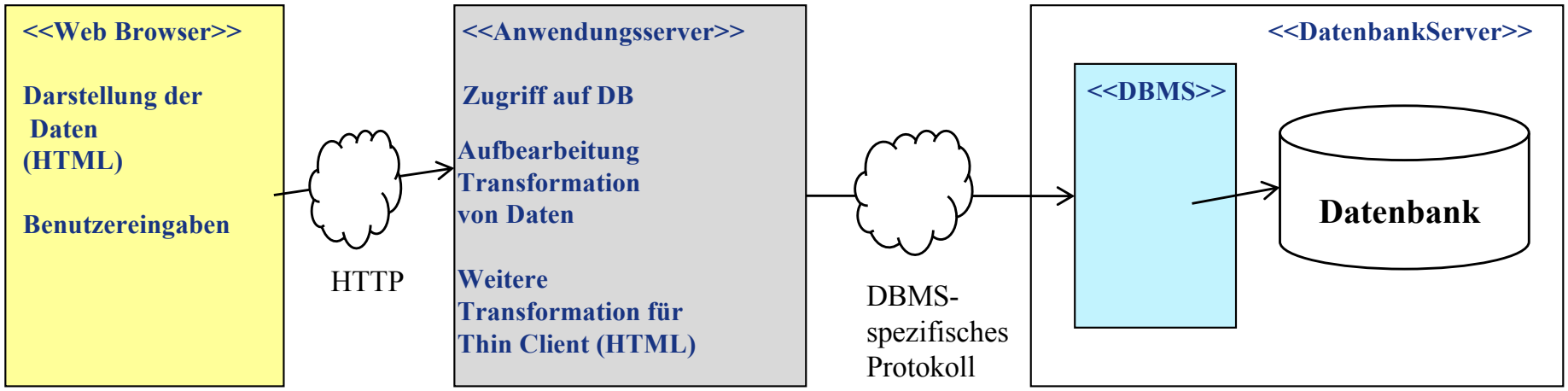
- In einer 3-Tier Architektur ist die Logik für die Darstellung von Daten auf dem Client, die Logik für den Zugriff, Transformation von Daten getrennt auf einem Anwendungserver



- Vorteil:
  - Bessere Trennung der Belange (Darstellung vs. Aufbearbeitung)
- Nachteil:
  - Komplexe Architektur, höherer Implementierungsaufwand
  - Schlechter für Szenarien mit hohem Datenvolumen

3-Tier Architektur mit Thin Client

- In einer 3-Tier Architektur ist die Logik für die Darstellung von Daten auf dem Client, die Logik für den Zugriff, Transformation von Daten getrennt auf einem Anwendungserver. Der Client ist ein Thin Client (z.B. Browser, Handy)



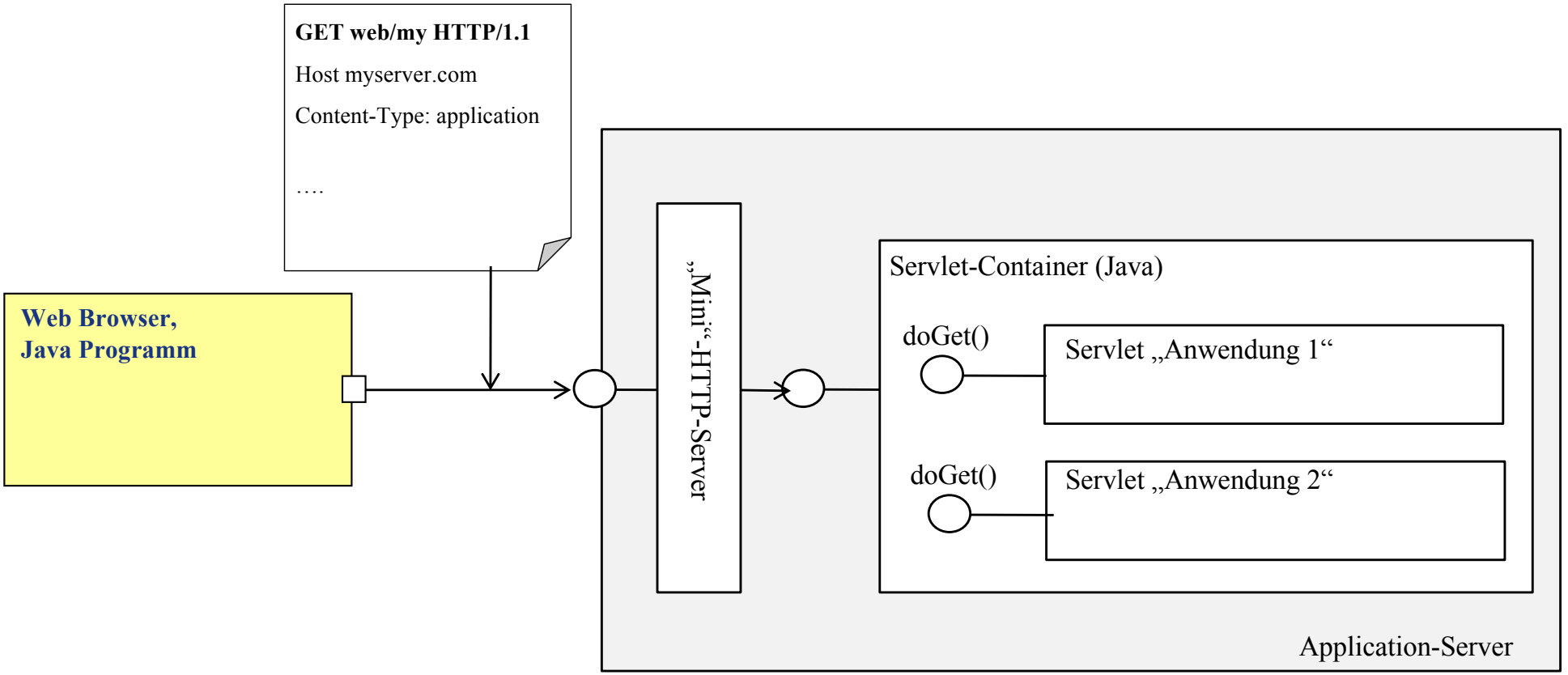
- Vorteil:
  - Client muss nicht unbedingt ein Java-Programm sein, Zugriff „von überall“ auf den Anwendungsserver
- Nachteil:
  - Komplexe Architektur, höherer Implementierungsaufwand
  - ggf. geringere Darstellungsmöglichkeiten

Exkurs: Servlet

- Ein Servlet ist eine Java-Klasse (package javax.servlet.*), mit der ein Java Programm ein HTTP-Request empfangen und verarbeiten kann.
  - Kapselung des HTTP-Requests in eine objektorientierte Struktur (Klasse HttpServletRequest)
  - Zentrale Methoden: doGet(), doPost() u.a.
  - Ausgabe von dynamischen HTML oder XML Code
- Zugriff auf JDBC-Klassen möglich

```
import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class HelloWorld extends HttpServlet {
  public void doGet(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
    PrintWriter out = response.getWriter();
    out.println("<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +
                "Transitional//EN\">\n" +
                "<html>\n" +
                "<head><title>Hello WWW</title></head>\n" +
                "<body>\n" +
                "<h1>Hello WWW</h1>\n" +
                "</body></html>");
  }
}
```

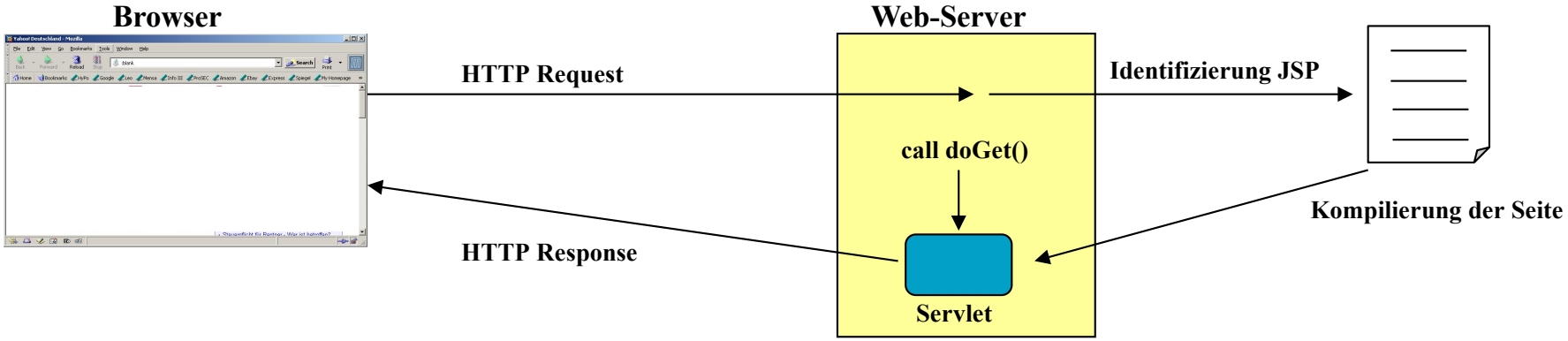
Exkurs: Servlet und Servlet Container



- Servlet-Klassen müssen in speziellen Laufzeitumgebungen (Servlet Container) installiert werden
- Servlet-Container nimmt einen HTTP-Request entgegen und kompiliert diesen in ein Java-kompatibles Format
  - Übergabe des Request an Methode doGet() bzw. doPost() eines selektierten Servlets (Bestimmung durch URL)

Exkurs: Java Server Pages

- Java Server Pages sind eine Erweiterung des Servlets-Modells, zur Vereinfachung der dynamischen HTML-Generierung: Java wird als Skriptsprache direkt in eine HTML-Seite einfügt (HTML+Java = JSP).
- Beim Aufruf einer JSP wird diese zu einem Servlet kompiliert:



- Nach der Kompilierung werden sämtliche Requests über das Servlet *direkt* behandelt
- Zugriff auf sämtliche Backend-Ressourcen (Datenbank) sowie auf das gesamte Java Spektrum.

Aufbau der Vorlesung

Kapitel 10: Java und Datenbanken - JDBC	
✓ 1	Motivation
✓ 2	Allgemeiner Überblick über JDBC
✓ 3	Entwicklung mit JDBC
✓ 4	Exkurs: Objektrelationales Mapping (ORM)
✓ 5	Exkurs: Realisierungsformen einer Datenbankanwendung in Java (N-Tier Architekturen)
6	**Zusammenfassung und Ausblick**

Zusammenfassung und Ausblick

- JDBC ermöglicht den einfachen Zugriff auf eine Datenbank
- Nicht so mächtig wie ein SQL-Client insbesondere bei der Auswertung einer Ergebnisrelation
  - Realisierung des Cursor-Prinzips
- Basis für die Entwicklung verschiedener N-Tier Architekturen
- Weiterführende Technologie notwendig, um die „Objekt-Relationen“ Lücke innerhalb einer Anwendung zu schließen
  - Objektrelationales Mapping (ORM)
  - Technologien wie z.B. Hibernate