HOCHSCHULE
FÜR ANGEWANDTE
WISSENSCHAFTEN · FH
MÜNCHEN

Integration von Datenbanken mit JDBC

Java Database Connectivity

- Ermöglicht Kommunikation zwischen Java-Anwendungen und Datenbanken
- Aktuelle Version: JDBC 4.2 (Java 1.8)
- Unabhängig vom verwendeten Datenbanksystem (write once, run anywhere)
- Tabellenbasiert (für relationale Datenbank Management Systeme DBMS)

Lieferantencode	Lieferantenname	Adresse
004	Baumgarten R.	Tankstr. 23
009	Strauch GmbH	Hintergarten 9

Grundlegender Funktionsumfang

Zugriff auf eine Datenbank um deren Ausprägung zu erfahren:

Verbindung aufbauen

↓

SQL-Anweisung absetzen

↓

Ergebnisse bearbeiten

↓

Metadaten ermitteln

Die Architektur

Packages:
java.sql.*

Java-Anwendung

JDBC API

JDBC Treibermanager

Verwaltung von JDBC-Treibern, Auswahl des korrekten Treibers → Datenbankunabhängigkeit

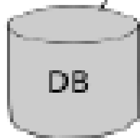
JDBC Treiber API

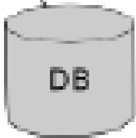
MySQL Treiber

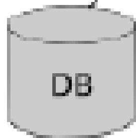
DB2 Treiber

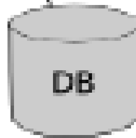
...

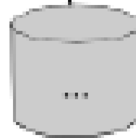
Übersetzen von JDBC-Aufrufen in spezifische Datenbankaufrufe











Die Treibertypen

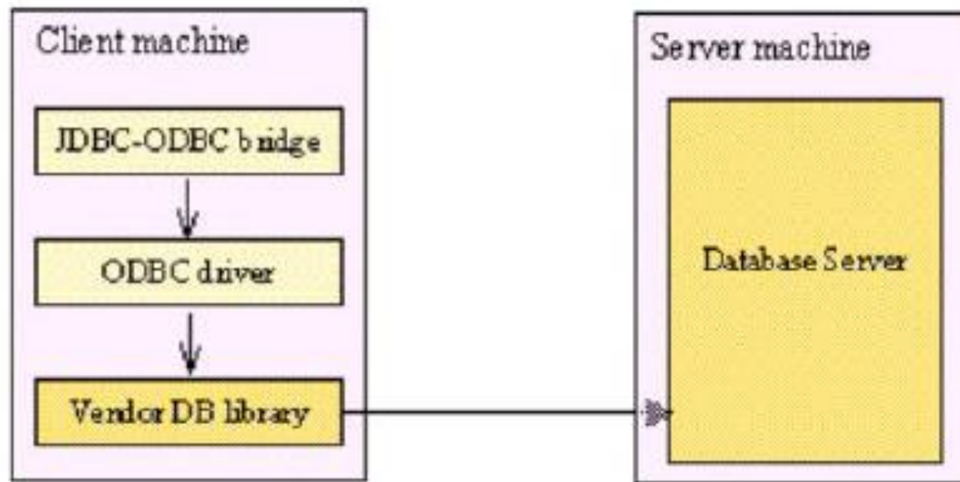
Es werden laut Oracle 4 Treibertypen definiert.

Unterscheidung anhand des nativen Anteils.

Native: Werden Aufrufe an eine Datenbankspezifische Bibliothek weitergeleitet oder nicht.

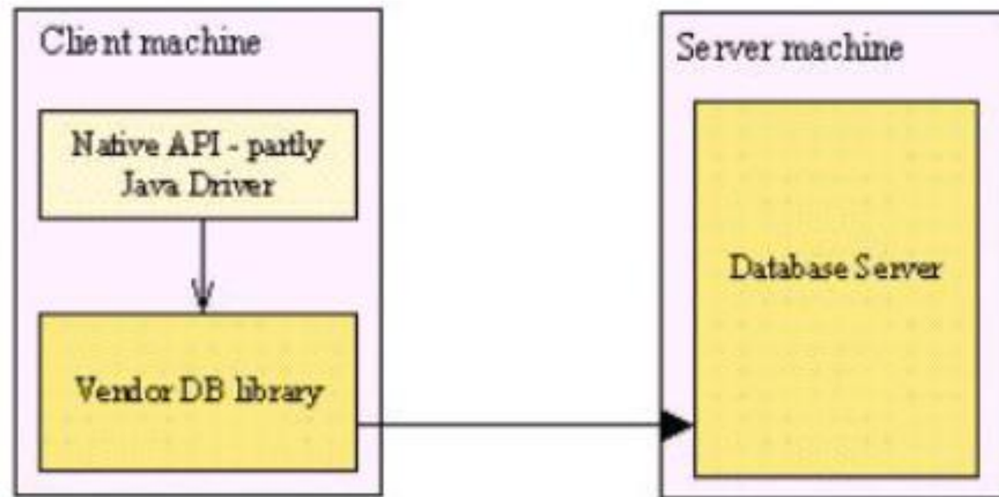
Typ 1 – JDBC-ODBC Brücke

- Übersetzt JDBC-Aufrufe in ODBC-Aufrufe
- Mit fast allen Datenbanksystemen verwendbar
- Kein Support mehr unter Java 8 (es existiert für jede Datenbank ein JDBC Treiber)
- Sehr langsam



Typ 2 – Native API Java Driver

- JDBC-Treiber verwendet datenbankspezifische API
- Plattformspezifische Datenbankbibliothek auf Client erforderlich (nicht portabel)
- Teils in einer anderen Sprache geschrieben (benötigt ODBC Zugriff)
- Schneller als Typ 1



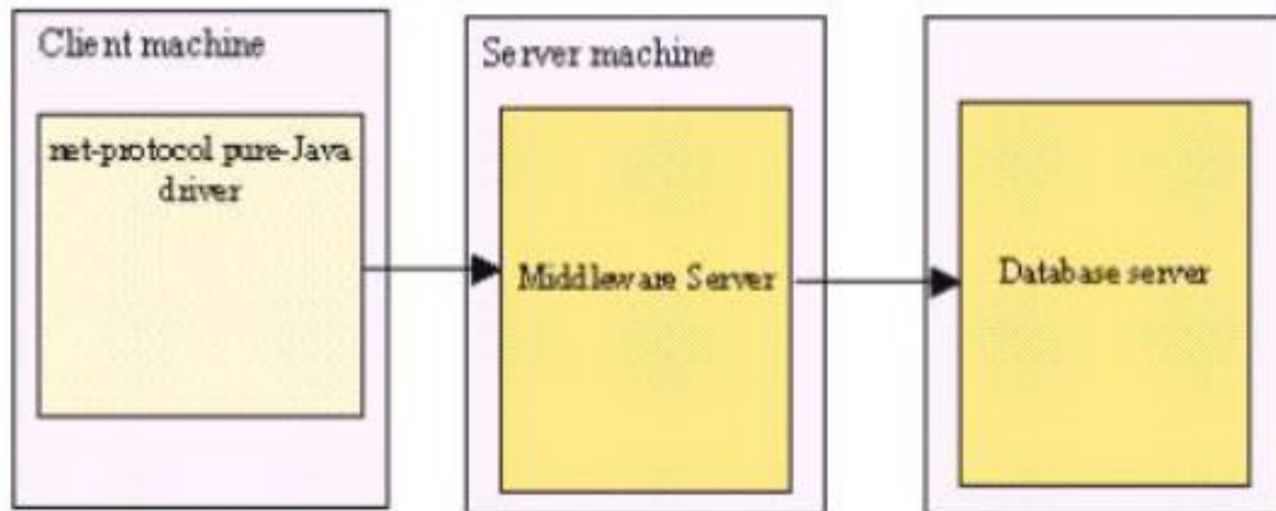
Thamm Philip, Fakultät 07, pthamm@hm.edu

Typ 3 – JDBC Net Driver

- 3 Schichten:
  - JDBC Aufrufe an Middleware
  - Middleware verwendet natives Datenbankprotokoll
- Kein Binärcode auf Client
- JDBC-API Befehle werden in generische DBMS-Befehle übersetzt und an die Middleware übermittelt
- Erst Middleware übersetzt Befehle zum jeweiligen Datenbankserver
- Muss nichts über den Datenbankserver wissen

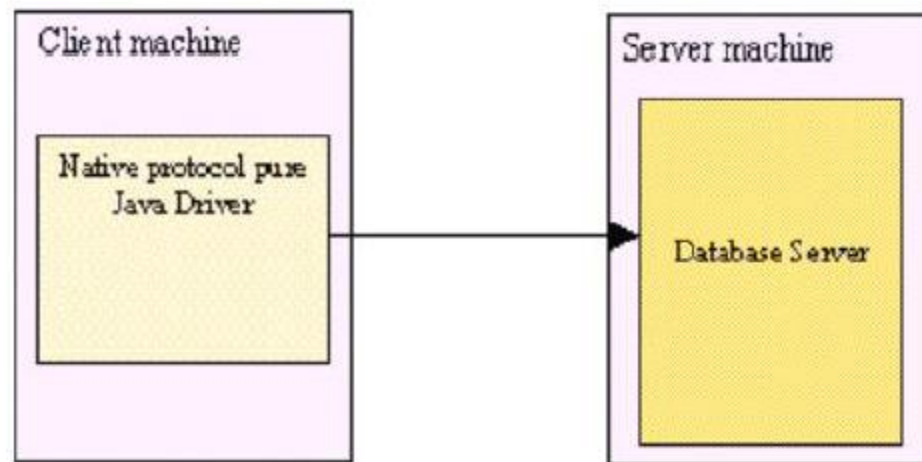
Typ 3 – JDBC Net Driver

- Hohes Optimierungspotential (Caching, Lastbalancierung)
- Verschlüsselung möglich



Typ 4 – Native Protocol Driver

- Direkte Kommunikation mit der Datenbank (Datenbankprotokoll)
- Javabasierte Treiber (plattformunabhängig)
- Schnell, aber weniger flexibel im Vergleich zu Typ 3



Thamm Philip, Fakultät 07, pthamm@hm.edu

Die JDBC-API

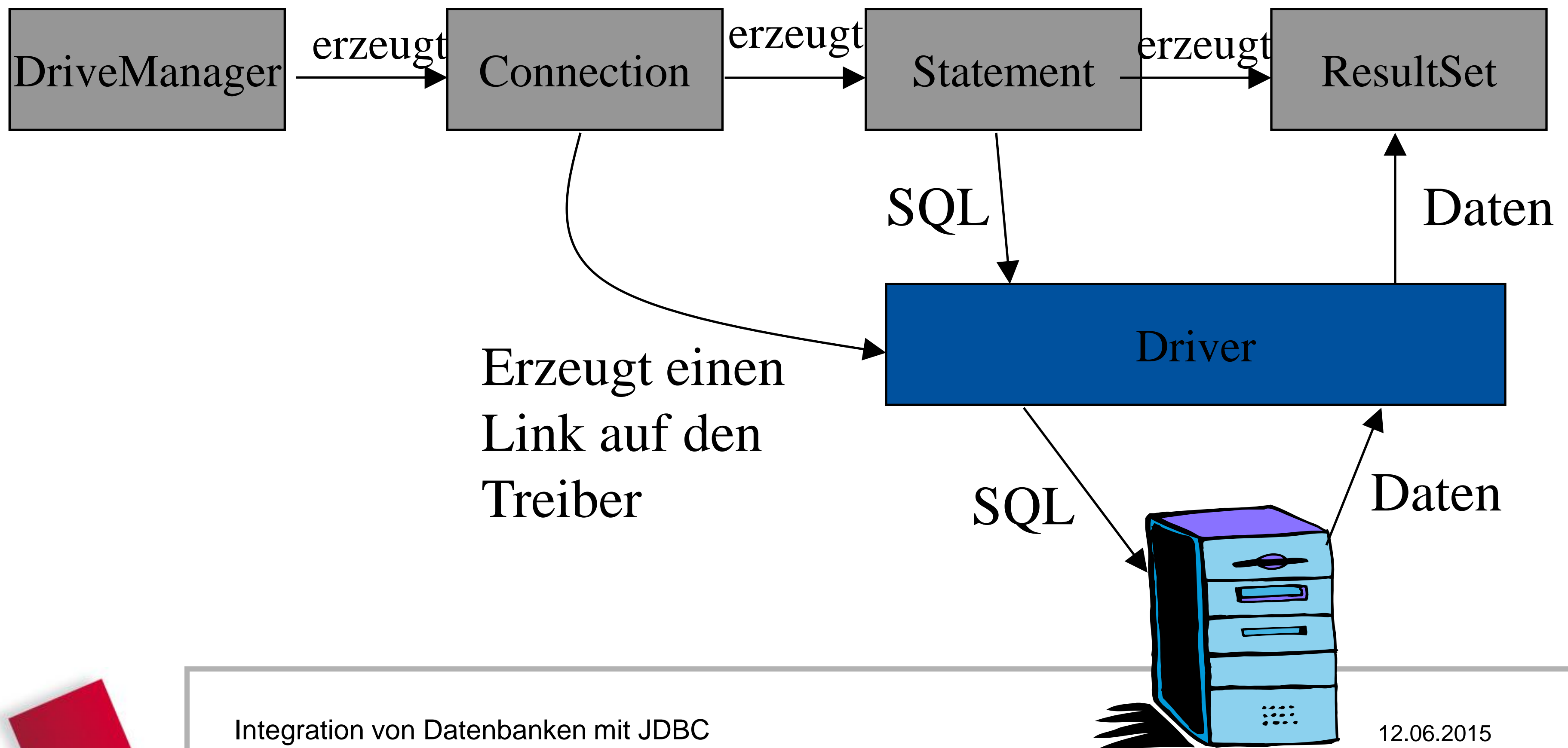
Pakete:

- java.sql.*

Wichtige Klassen:

- **DriverManager**: Schnittstelle zur JDBC-API
- **Connection**: Verbindung zu einer Datenquelle
- **Statement**: Abarbeiten von SQL-Anweisungen
- **PreparedStatement**: vorkompilierte SQL-Anweisung
- **CallableStatement**: vorkompilierte Stored-Procedure-Aufrufe
- **ResultSet**: Verwaltet Ergebnisse einer Abfrage
- **SQLException**: Informationen im Fehlerfall

JDBC-API Grundlegender Ablauf



Schritte zur Datenbankanbindung

1. Einbinden der JDBC-Datenbanktreiber in den Klassenpfad
2. Unter Umständen Anmelden der Treiberklassen
3. Verbindung zur Datenbank aufbauen
4. Eine SQL Anweisung erzeugen
5. Die SQL Anweisung ausführen
6. Das Ergebnis/die Ergebnismenge der Anweisung holen und verarbeiten
7. Die Datenbankverbindung schließen

JDBC benutzen

Laden des Treibers:

- Der Treiber-Klassenpfad muss seit Java 8 dem Java Compiler nicht mehr bekannt gemacht werden.

Eine Verbindung aufbauen

Eine Verbindungs-URL benötigt das Literal jdbc: gefolgt vom Namen des Treibers der Datenbank und der URL zur Datenbank

Erzeugen eines Connection-Objektes

```
// STEP 3 : Open a connection
System.out.println("Connecting to database...");
Connection con = DriverManager.getConnection("jdbc:mysql://localhost/sakila2", USER, PASS);
```

JDBC benutzen (Fortsetzung)

Erzeugen eines Statement-Objektes

- Ausführen eines Statements & Abfragen des Ergebnisses
- Explizites Schließen notwendig (stmt.close())
- Ausführen einer Anweisung verwirft bisherige Ergebnisse
- Ergebnis vom Typ ResultSet oder Änderungszähler

```
// Eine SQL Anweisung erzeugen
System.out.println("Creating statement...");

String sql;
    sql = "SELECT actor_id, last_name, last_update from actor ORDER BY last_name";

// Die SQL Anweisung ausführen
stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(sql);
```

JDBC benutzen (Fortsetzung)

Wichtige Methoden

- ResultSet executeQuery(String sql)
  - Ausführen einer Anweisung
- Int executeUpdate(String sql)
  - Ausführen einer Änderung
- Boolean execute(String sql)
  - Ausführen einer beliebigen Anweisung
- Int getUpdateCount()
  - Anzahl geänderter Tupel
- ResultSet getResultSet()
  - Liefert aktuelles Ergebnis, falls ResultSet (sonst null)

JDBC benutzen (Fortsetzung)

ResultSet-Objekt

- Repräsentiert Anfrageergebnisse (Tabelle)
- Grundprinzip: Cursor (Iterator)
  - Zugriff nur auf aktuelle **Zeile**
  - Boolean next()

    positioniert Cursor auf nächster Zeile

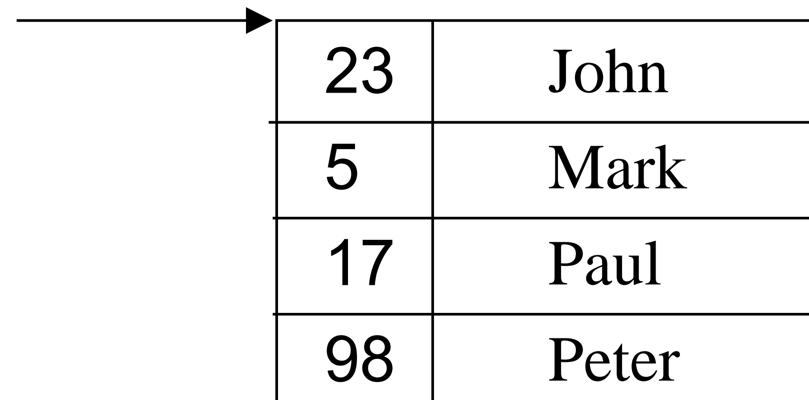
    ermittelt ob weitere Tupel verfügbar
- Explizietes Schließen notwendig (close())

JDBC benutzen (Fortsetzung)

ResultSet Verarbeiten:

- Cursor Operationen
  - first(), last(), next(), previous()
- Arten der Durchsuchung
  - Forward only: sequentielle Abarbeitung
  - Scrollable: freie Positionierung des Cursors (aufwändiger!)
- Typischer Code

```
while( rs.next() ) {
  // process the row;
}
```



JDBC benutzen (Fortsetzung)

ResultSet Beispiel

```
// Verarbeiten von Ergebnissen
while (rs.next()) {
    // Retrieve by column name
    int actor_id = rs.getInt("actor_id");
    String last_name = rs.getString("last_name");
    Date last_update = rs.getDate("last_update");

    // Display values
    System.out.print("actor_id: " + actor_id);
    System.out.print(", last_name: " + last_name);
    System.out.print(", last_update: " + last_update + "\n");
}
```

JDBC benutzen (Fortsetzung)

Vorkompilierte SQL-Anweisungen

- Kompilieren kostet Zeit
- Idee: oft verwendete Anweisungen nur einmal kompilieren

  -> **PreparedStatement**
- Erzeugen mittels: conn.preparedStatement(String sql)
- Unterklasse von Statement
- Parametrisierbar mittels „?“

```
PreparedStatement pstmt = conn.prepareStatement("Select salary from employee where Lastname =?");
pstmt.setString(1, "Testname");
ResultSet rst = pstmt.executeQuery();
rst.next();
System.out.println(rst.getDouble(1));
rst.close();
pstmt.close();
```

JDBC benutzen (Fortsetzung)

CallableStatement Objekt:

- Ausführen von Stored-Procedures des Datenbanksystems
- Erzeugen mittels con.prepareCall(String sql)
  - Parametersyntax „{CALL <name> (?,…) }“
  - Automatisches Umschreiben auf Syntax des DBMS
- Unterklasse von PreparedStatement
- Ausgabeparameter müssen registriert werden

JDBC benutzen (Fortsetzung)

Beispiel zu CallableStatements (Stored Procedures):

```
CallableStatement cs = conn.prepareCall("{CALL GET_A_TABLE(?,?)}");
cs.setString(1, "EMPLOYEE");
cs.registerOutParameter(2, Types.VARCHAR);
cs.execute();
// Prints select * from EMPLOYEE
System.out.println("Parameter 2 (SQL): " + cs.getString(2));
while(rs.next())
    // Prints the LASTNAME
    System.out.println(rs.getString("LASTNAME"));
rs.close();
cs.close();
```

JDBC benutzen (Fortsetzung)

Verbindung auflösen und aufräumen

- Nach erfolgreicher Durchführung
- ResultSet nach durchlaufen der Schleife
- Statement am Ende der Methode
- Connection beim beenden der Anwendung
- Connection wird für die gesamte Benutzungsdauer offen gehalten

```
// STEP 7 : Verbindungen auflösen und aufräumen
rs.close();
stmt.close();
conn.close();
```

JDBC-Abstraktion in Spring

```
onnection conn = null;
tatement stmt = null;
esultSet rs = null;
ry {
   // STEP 3 : Open a connection
   System.out.println("Connecting to database...");
   conn = DriverManager.getConnection(DB_URL, USER, PASS);

   // Die SQL Anweisung ausführen
   stmt = conn.createStatement();
   rs = stmt.executeQuery(sql);

   // Verarbeiten von Ergebnissen
   while (rs.next()) {
   }
   // STEP 7 : Verbindungen auflösen und aufräumen
   rs.close();
   stmt.close();
   conn.close();
 catch (SQLException se) {
   // Handle errors for JDBC
   se.printStackTrace();
 // end try
inally {
   if (rs != null)
       try {
           rs.close();
       } catch (SQLException e) {
           e.printStackTrace();
       }
   if (stmt != null)
       try {
           stmt.close();
       } catch (SQLException e) {
           e.printStackTrace();
       }
```

Umfangreiche try/catch Blöcke

Konkrete Angabe der Datenbankinformation

Selbstständiges Kümmern um die Verbindung

Finally im Fehlerfall

Architekturelle Übersicht Spring

Spring AOP
Source-level metadata
AOP infrastructure

Spring ORM
Hibernate support
iBatis support
JDO support

Spring DAO
Transaction infrastructure
JDBC support
DAO support

Spring Web
WebApplicationContext
Multipart resolver
Web utilities

Spring Context
Application context
UI support
Validation
JNDI, EJB support & Remoting
Mail

Spring Web MVC
Web MVC Framework
Web Views
JSP / Velocity
PDF / Excel

Spring Core
Supporting utilities
Bean container

JDBC-Abstraktion in Spring

+ Liefert eine neue durchsichtigere Exception hierarchie
+ Enthält das JdbcTemplate mit vielen komfortablen Methoden für einen leichteren Datenbankzugriff
+ Enthält einen Objektlayer zusätzlich zum JdbcTempate. Dieser Layer enthält folgende Klassen SqlQuery, SqlUpdate and StoredProcedure für einen "objektorientierten" Ansatz
+ Kümmert sich um die Datenbankverbindung
  (kein try/catch)
+ Reduziert den Code der geschrieben werden muss

JDBC-Abstraktion in Spring

Spring kann für alle denkbaren Java-Applikationen wertbringend sein.

Spring kümmert sich in JDBC um:

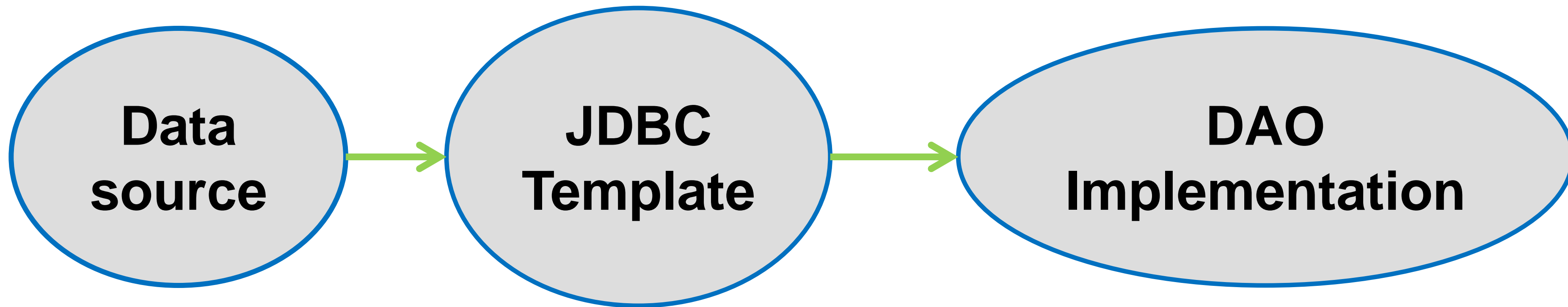
Task	Spring	You
Connection Management	√	
SQL		√
Statement Management	√	
ResultSet Management	√	
Row Data Retrieval		√
Parameter Declaration		√
Parameter Setting	√	
Transaction Management	√	

JDBC-Abstraktion in Spring

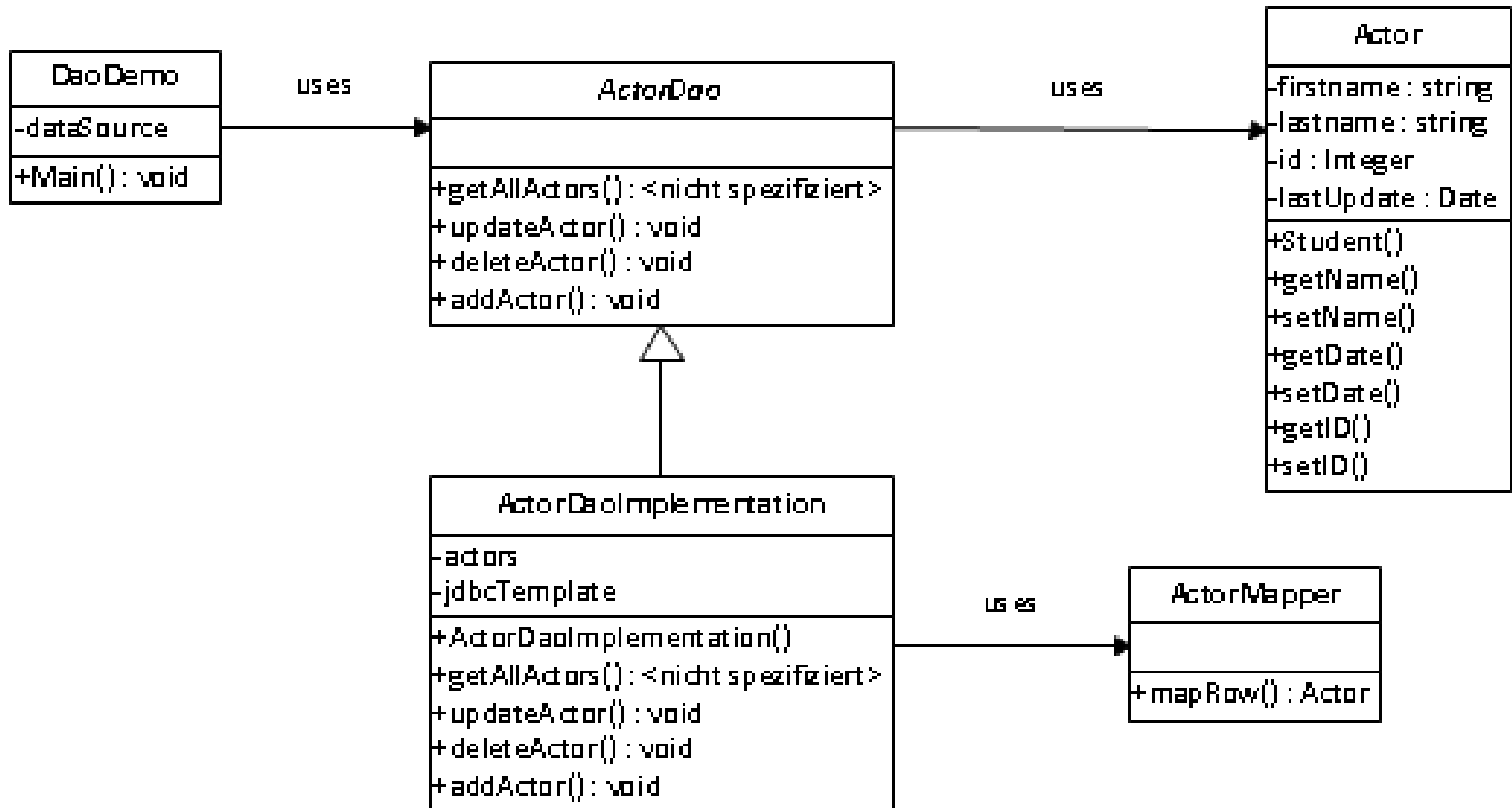
Allen voran steht in Spring die Klasse **JDBCTemplate**:

- Führt SQL Anweisungen aus
- Liest Ergebnisse ein
- Ermöglicht mit Hilfe des RowMappers Resultate zu ermitteln
- Mapt Spalten auf Objekte

JDBCTemplate Dependencies



DAO Pattern



JDBC-Abstraktion in Spring

Beans.xml Konfiguration:

- Definieren der Datenbankverbindung
- Es können verschiedene Datenbankverbindungen definiert werden
- Ansprechen über so genannte DataSources als Dependency
- Lookup über JNDI

```
<!-- Initialization for data source -->
<bean id="dataSource"
   class="org.springframework.jdbc.datasource.DriverManagerDataSource">
     <property name="driverClassName" value="com.mysql.jdbc.Driver" />
     <property name="url" value="jdbc:mysql://localhost:3306/sakila" />
     <property name="username" value="root" />
     <property name="password" value="" />
 </bean>
```

JDBC-Abstraktion in Spring

Mit Hilfe des DAO (Data Access Object)-Patterns werden Spalten auf Objekte gemapt

Die Klasse (**Actor**) sollte Attribute äquivalent zu denen der Tabelle besitzen.

```
// Ein Schauspieler
public class Actor {

	private int actor_id;
	private String last_name;
	private Date last_update;

	public Actor(int actor_id, String last_name, Date last_update) {
		this.actor_id = actor_id;
		this.last_name = last_name;
		this.last_update = last_update;
	}
```

JDBC-Abstraktion in Spring

Mit Hilfe einer **RowMapper** Klasse werden dann Abfrageergebnisse auf einzelne Objekte gemapt. Diese Klasse erhält vom Spring-Framework für jede Zeile ein ResultSet. Dieses wird dann gemapt.

```
// Implementiert das Interface RowMapper
public class ActorMapper implements RowMapper<Actor> {

    @Override
    // Erhält vom Spring-Framework einen Eintrag und mapt diesen auf ein Objekt
    public Actor mapRow(ResultSet rs, int rowNum) throws SQLException  {
        return new Actor(rs.getInt("actor_id"), rs.getString("last_name"),
                rs.getDate("last_update"));
    }

}
```

JDBC-Abstraktion in Spring

Ausführung des Programms (Main):

- Zu erst muss sich die DataSource (Datenbankverbindung) der Beans.xml besorgt werden

```
// Liefert die in der Beans.xml konfigurierte DataSource
private static DataSource getDataSource() {
    // Pfad zu der Konfigurationsdatei
    ApplicationContext ac = new ClassPathXmlApplicationContext("Beans.xml");
    // Bean mit der ID 'mysqlDataSource' (MySQL Konfiguration) laden
    DataSource dataSource = (DataSource) ac.getBean("dataSource");

    // Beispiel einer hardcodierten DataSource
    // DriverManagerDataSource dataSource2 = new DriverManagerDataSource();
    // dataSource2.setDriverClassName("com.mysql.jdbc.Driver");
    // dataSource2.setUrl("jdbc:mysql://localhost:3306/sakila2");
    // dataSource2.setUsername("root");
    // dataSource2.setPassword("");

    return dataSource;
}
```

JDBC-Abstraktion in Spring

Ausführung des Programms (Main):

- Definieren des SQL Statements sowie mappen der Ergebnisse auf Objekte
- Man erhält aus dem JDBCTemplate direkt eine Liste der Objekte

```
// Liefert ein mit der vorkonfigurierten DataSource gefülltes
// JDBCTemplate
DataSource dataSource = getDataSource();
JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

// SQL Query
String sql = "SELECT actor_id, last_name, last_update from actor ORDER BY last_name";

List<Actor> actorList = jdbcTemplate.query(sql, new ActorMapper());

// Ausgeben aller Schauspieler
for (Actor actor : actorList) {
    System.out.println(actor);
}
```

Einbindung von JDBC in Java EE Anwendungen

- Java-EE Anwendungen benötigen ohnehin einen Applikationsserver
- Der Datenbanktreiber muss am Applikationsserver eingebunden werden
- Es muss eine Datenquelle (DataSource) am Applikationsserver eingerichtet werden
- Referenzieren der Datenquelle über JNDI (**J**ava **N**aming and **D**irectory **I**nterface)
- Die Datenquelle kann im Quelltext Injected werden
- Abfragen der Datenbank wie im normalen JDBC

Thamm Philip, Fakultät 07, pthamm@hm.edu

Einbindung von JDBC in Java EE Anwendungen

Java Applikation muss lediglich JNDI Namen kennen.

Folgende Punkte muss sie **nicht** kennen:

- Den konkreten Datenbankserver
- Benutzername für die Datenbank
- Kennwort für die Datenbank

Einbindung von JDBC in Java EE Anwendungen

Datenquelle am Applikationsserver

Home Deployments Configuration Runtime Administration

Subsystems

- Connector
  - JCA
  - Datasources
  - Resource Adapters
  - Mail
- Container
- Core
- Infinispan
- Security
- Web

General Configuration

- Interfaces
- Socket Binding
- Paths
- System Properties

DATASOURCES XA DATASOURCES

Name	JNDI	Ena
ExampleDS	java:jboss/datasources/ExampleDS	✔
fwpFach	java:/jdbc/name=fwpFach	✔

Attributes Connection Pool Security Properties Validation Timeouts

Edit

Name: fwpFach

JNDI: java:/jdbc/name=fwpFach

Der JNDI Name

Statistics enabled?: false

Driver: mysql-connector-java-5.1.35-bin.jar_com.mysql.jdbc.Driver_5_1

Der Treiber

Einbindung von JDBC in Java EE Anwendungen

Datenquelle am Applikationsserver

DATASOURCES | XA DATASOURCES

Attributes | Connection | Pool | Security | Properties | Validation | Timeouts

Test Connection

Need Help?

Connection URL:	jdbc:mysql://127.0.0.1:3306/sakila2
New Connection Sql:	
Transaction Isolation:	
Use JTA?:	true
Use CCM?:	true

Die Datenbank Verbindung

Einbindung von JDBC in Java EE Anwendungen

Injecten der Datenquelle im Quelltext

```
// DataSource für den SQL Zugriff über eine Annotation injecten
@Resource(lookup = "java:jboss/datasources/fwpFach")
private DataSource dataSource;
```

Einbindung von JDBC in Java EE Anwendungen

Abarbeiten der SQL Anweisung wie in Beispiel 1

Umlenken der Ausgabe aus dem Servlet in HTML Quelltext

```
protected void doGet(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException {
    PrintWriter out = response.getWriter();
    out.println("<h1>Actor Ausgabe:</h1>");

    // Verwenden der SQL DataSource
    // Abhandlung wie Beispiel 1
    try (Connection connection = dataSource.getConnection();
            PreparedStatement preparedStatement = connection
                    .prepareStatement("SELECT * FROM actor");
            ResultSet resultSet = preparedStatement.executeQuery();) {
        while (resultSet.next()) {
            String lastName = resultSet.getString("last_name");
            String firstName = resultSet.getString("first_name");
            out.println("Actor: " + lastName + " " + firstName + "<br>");
        }
    } catch (SQLException e) {
        throw new IllegalStateException("Failed to fetch Actors", e);
    }
```

Verwenden der DS

Fazit

JDBC:

- ➕ JDBC Standard zum Zugriff auf Datenbanken aus Java heraus (kein Weg führt vorbei)
- ➕ Datenbankunabhängig / Plattformunabhängig
- ➕ Wichtige Klassen Connection, Statement, ResultSet
- ➖ Viel try/catch → finally
- ➖ Permanentes Überprüfen der Datenverbindung (noch offen?)

Spring:

- ➕ Spring deutlich komfortabler und handlicher
- ➕ Kein try/catch mehr
- ➕ Enheitliches Exceptionhandling
- ➕ Kein Verbindungsmanagement

Fazit

Java EE:

- Java EE aufwändig wie native JDBC
+ Kombination mit Spring möglich

  jdbcTemplate kann ein Datasource Objekt übernehmen z.B. eines vom Wildfly Server

```
jdbcTemplate = new JdbcTemplate(dataSource);
```

+ Elegantere Lösung mittels JPA möglich

Vielen Dank