Datenbanken und Informationssysteme

Programmieren mit Datenbanken

Burkhardt Renz

Fachbereich MNI
TH Mittelhessen

Sommersemester 2020

Übersicht

- Konzepte des Datenbankzugriffs
  - Programmierschnittstellen von SQL-Datenbanken
  - Ändern von Daten in der Datenbank
- JDBC
- ADO.NET
- Objekt-relationales Mapping mit JPA

Fragestellungen

1. Wie verwenden wir die Funktionalität des DBMS in unserem Anwendungsprogramm?
2. Wie kommen die Ergebnisse einer SQL-Anweisung in die Variablen unseres Programms?
3. Wie bekommen wir Informationen über die Ergebnisse einer SQL-Anweisung?
4. Wie können wir Daten in der Datenbank verändern?

Frage 1

Wie verwenden wir die Funktionalität des DBMS in unserem Anwendungsprogramm?

1. SLI – Statement-Level-Interface
2. CLI – Call-Level-Interface

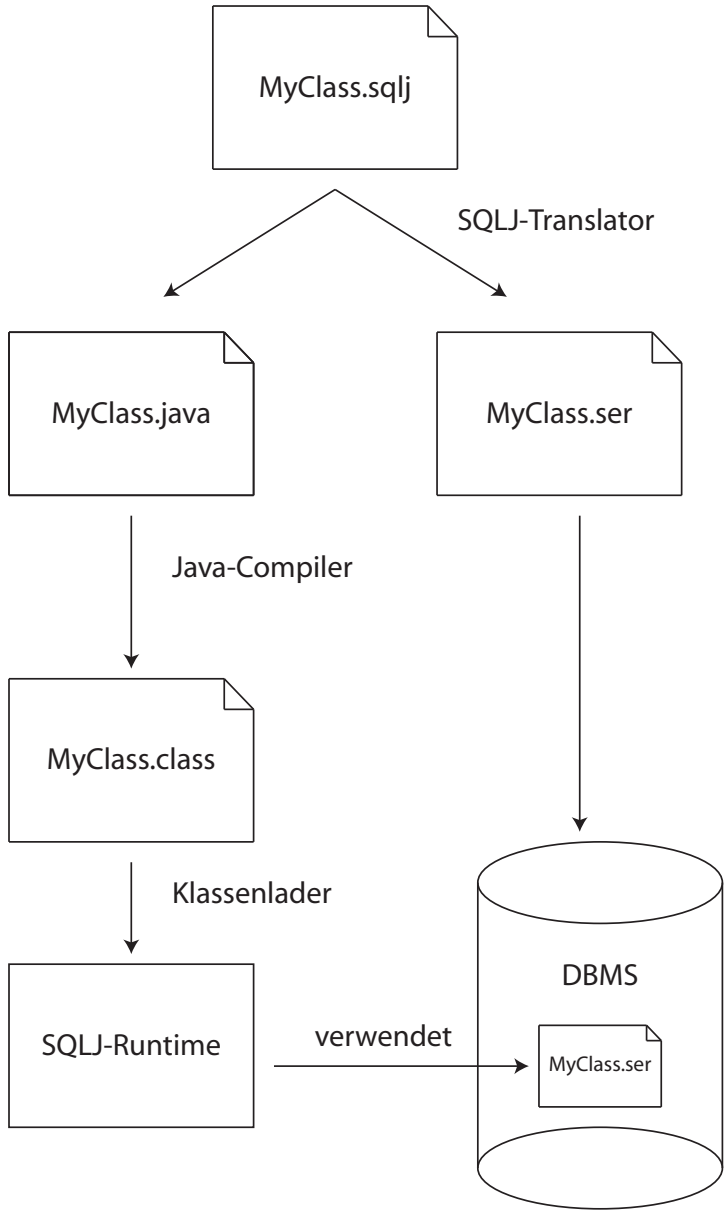
SLI am Beispiel von SQLj I

```
public class BookReader {
  public static void main(String[] args) {
    try {
      connect();
      readBook( "3-257-21755-2" );
    } catch (SQLException e) {
      e.printStackTrace();
    }
  }

  public static void readBook( String isbn ) throws SQLException {

  // siehe folgende Seite
  }
}
```

SLI am Beispiel von SQLj II

```
public static void readBook( String isbn ) throws SQLException {
  String author = null;
  String title = null;
  #sql {
      select author, title into :author, :title
        from books where isbn = :isbn};
  System.out.println(author + ": " + title);
}
```

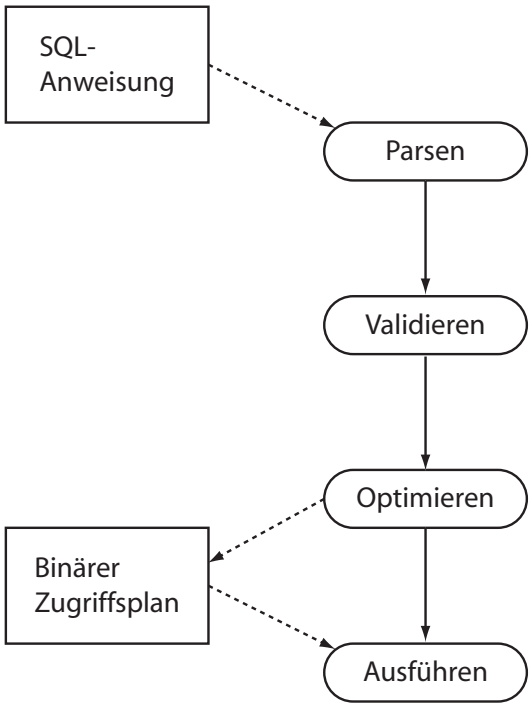
Konzept von SQLj

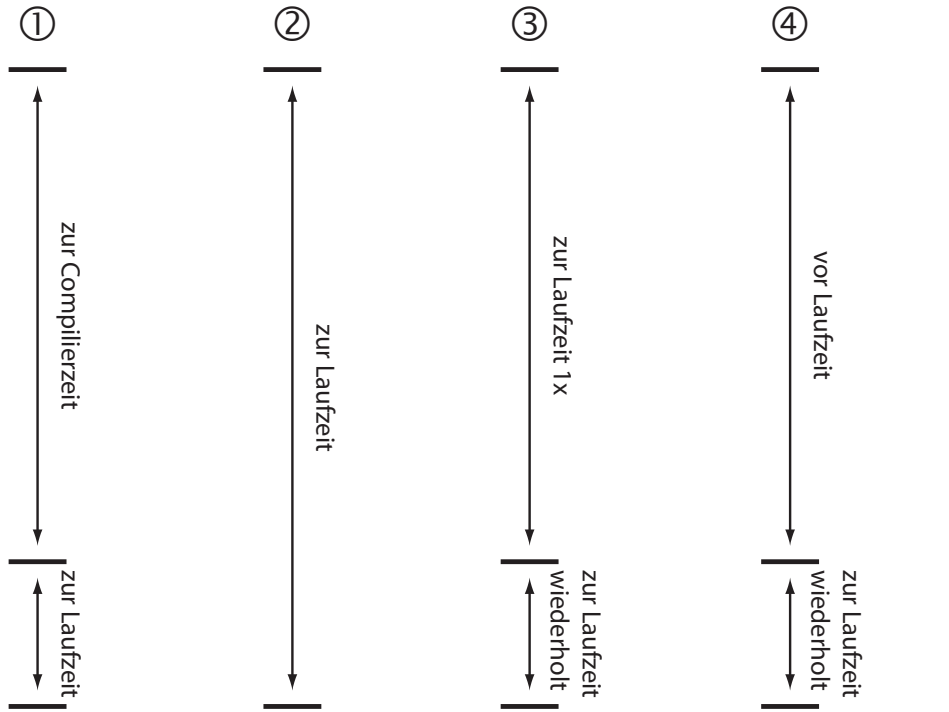


CLI am Beispiel von JDBC

```
public class BasicJDBC {

  public static void main(String[] args) {

  // ... Erstellen einer Connection con

    stmt = con.createStatement();

    rs = stmt.executeQuery("select author, title from Books");

    while (rs.next()) {
      System.out.println(rs.getString("author") + " "
         + rs.getString("title"));

  // ...
  }
}
```

Verarbeitung einer SQL-Anweisung





statisches SQL
eSQL

dynamisches SQL
EXECUTE IMMEDIATE
in eSQL
Statement
in JDBC

dynamisches SQL
PREPARE/EXECUTE
in eSQL
PreparedStatement
in JDBC

Stored Procedure
CALL
in eSQL
CallableStatement
in JDBC

Frage 2

Wie kommen die Ergebnisse einer SQL-Anweisung in die Variablen unseres Programms?

1. Cursorkonzept
2. JDBC: Interface *ResultSet*

Arten von Cursorn

1. bzgl. der Sichtbarkeit von Änderung
   INSENSITIVE, SENSITIVE, ASENSITIVE
2. bzgl. der Navigierbarkeit
   FORWARD_ONLY, scrollable
3. bzgl. Änderbarkeit via Cursor
   READ_ONLY, UPDATABLE

Frage 3

Wie bekommen wir Informationen über die Ergebnisse einer SQL-Anweisung?

1. Indikatorvariable
2. SQL-Deskriptoren

Beispiele für Metadaten in JDBC

1. `select Verlag from Books`
   – Was passiert, wenn Verlag <null> ist?
2. `select * from Books`
   – Welchen Aufbau hat die Ergebnismenge?

1. Spezielle Methode in JDBC
   `ResultSet::wasNull()`
2. Spezielles Objekt in JDBC
   `ResultSetMetaData` erzeugt via
   `ResultSet::getMetaData()`

Frage 4

Wie ändern wir Daten in der Datenbank?

1. Deklarative Änderung („searched update“)
2. Änderung mit Verwendung eines Cursors („positioned update“ – navigierende Änderung)

Umsetzung der Techniken

1. Änderung ohne Cursor

```
#sql{
  insert into Books(ISBN, Author, Title)
    values( :isbn, :author, :title) };
```

bzw.

```
stmt.executeUpdate( "insert into ..." );
```

2. Mit Verwendung eines Cursors

```
#sql myCursor = { select ... from ...};
...
#sql {update Books set Verlag = :verlag where current of :myCursor};
```

bzw.

```
rs.next();
rs.updateString( "Verlag", "neuer Verlag" );
rs.updateRow();
```

Übersicht

- Konzepte des Datenbankzugriffs
- JDBC
  - Grundlegendes Beispiel
  - Architektur von CLI am Beispiel JDBC
  - Wichtige Methoden
- ADO.NET
- Objekt-relationales Mapping mit JPA

Grundlegendes Beispiel in JDBC I

```
import java.sql.*;

public class BasicJDBC {

  public static void main(String[] args) {

    Connection con = null;
    Statement stmt = null;
    ResultSet rs   = null;
```

Grundlegendes Beispiel in JDBC 2

```
try {

   /** Schritt 1: JDBC-Treiber registrieren */
   Class.forName("org.postgresql.Driver");

   /** Schritt 2: Connection zum Datenbanksystem herstellen */
   con = DriverManager.getConnection(
              "jdbc:postgresql://localhost/azamon", "dis", "ChrisDate");

   /** Schritt 3: Statement erzeugen */
   stmt = con.createStatement();
```

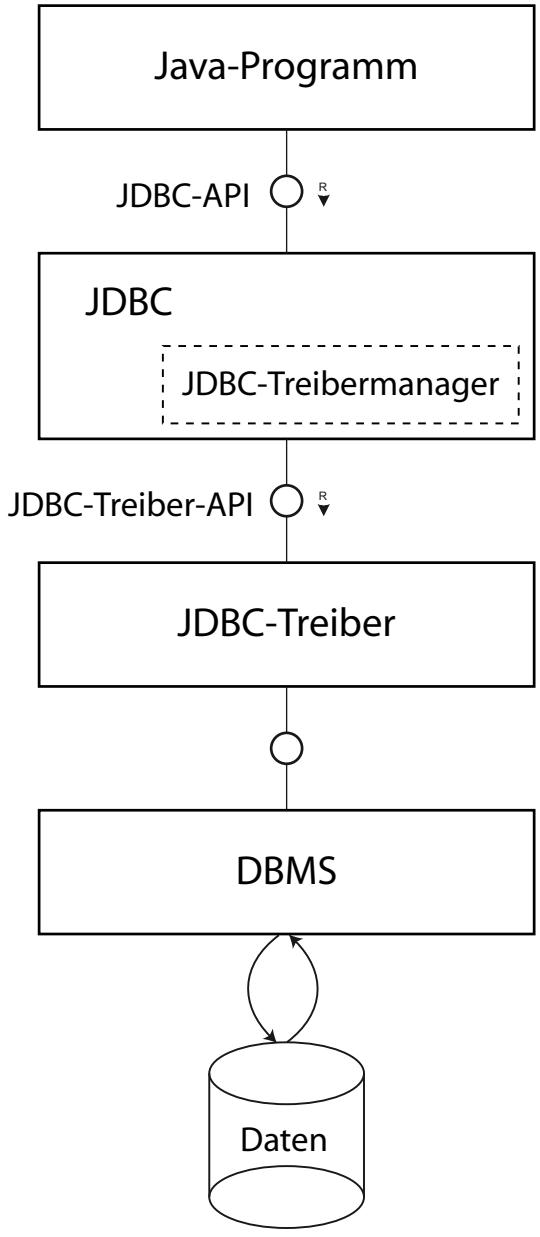
Grundlegendes Beispiel in JDBC 3

```
  /** Schritt 4: Statement direkt ausführen */
  rs = stmt.executeQuery("select author, title from Books");

  /** Schritt 5: Ergebnis der Anfrage verwenden */
  while (rs.next()) {
    System.out.println(rs.getString("author") + " "
        + rs.getString("title"));
  }
} catch (Exception e) {
  System.out.println(e.getMessage());
} finally {
```

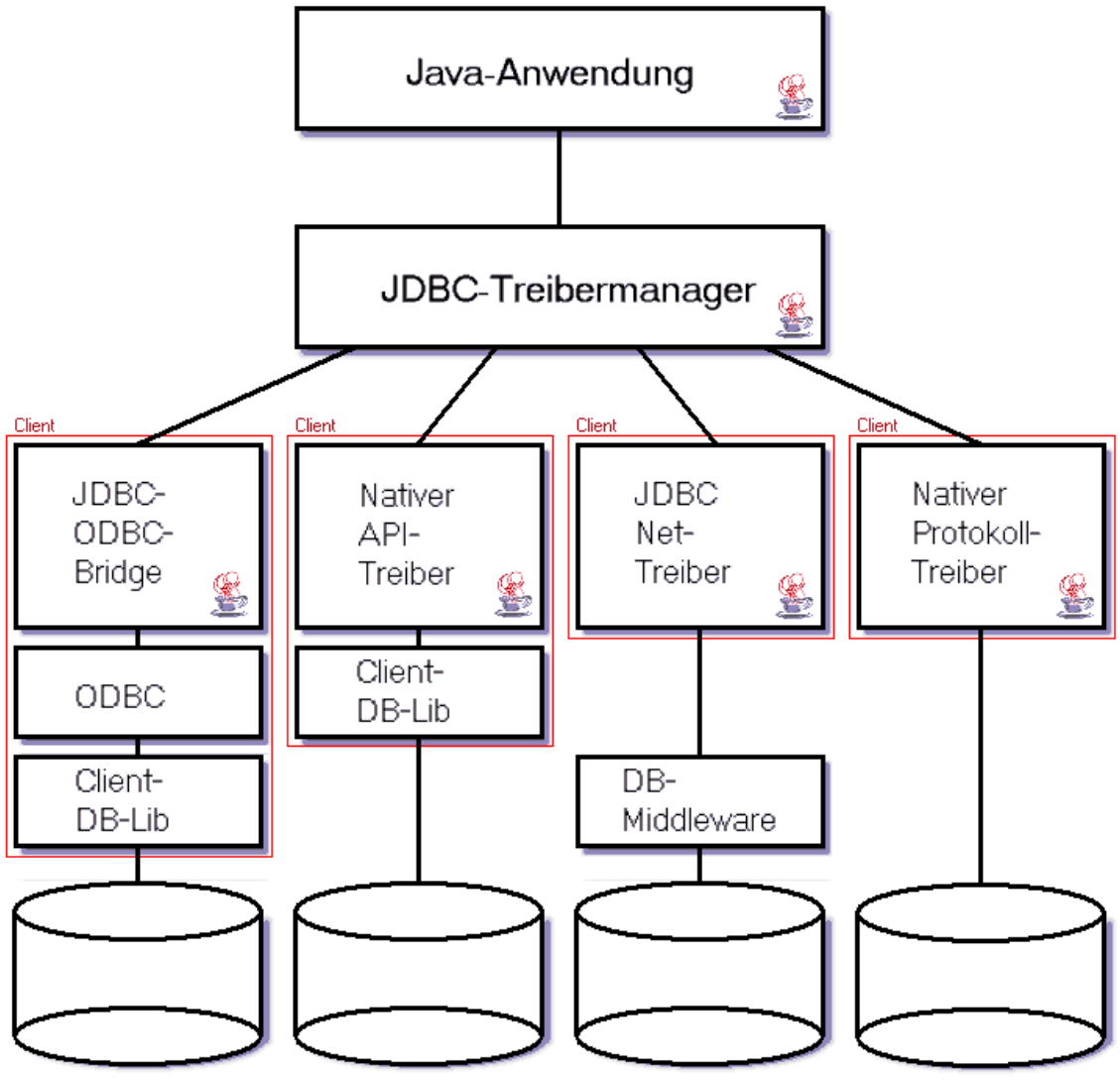
Grundlegendes Beispiel in JDBC 4

```
        /** Schritt 6: Ressourcen freigeben */
        try {
          if (rs != null) rs.close();
          if (stmt != null) stmt.close();
          if (con != null) con.close();
        } catch (Exception e) {
          System.out.println(e.getMessage());
        }
      }
    }
}
```

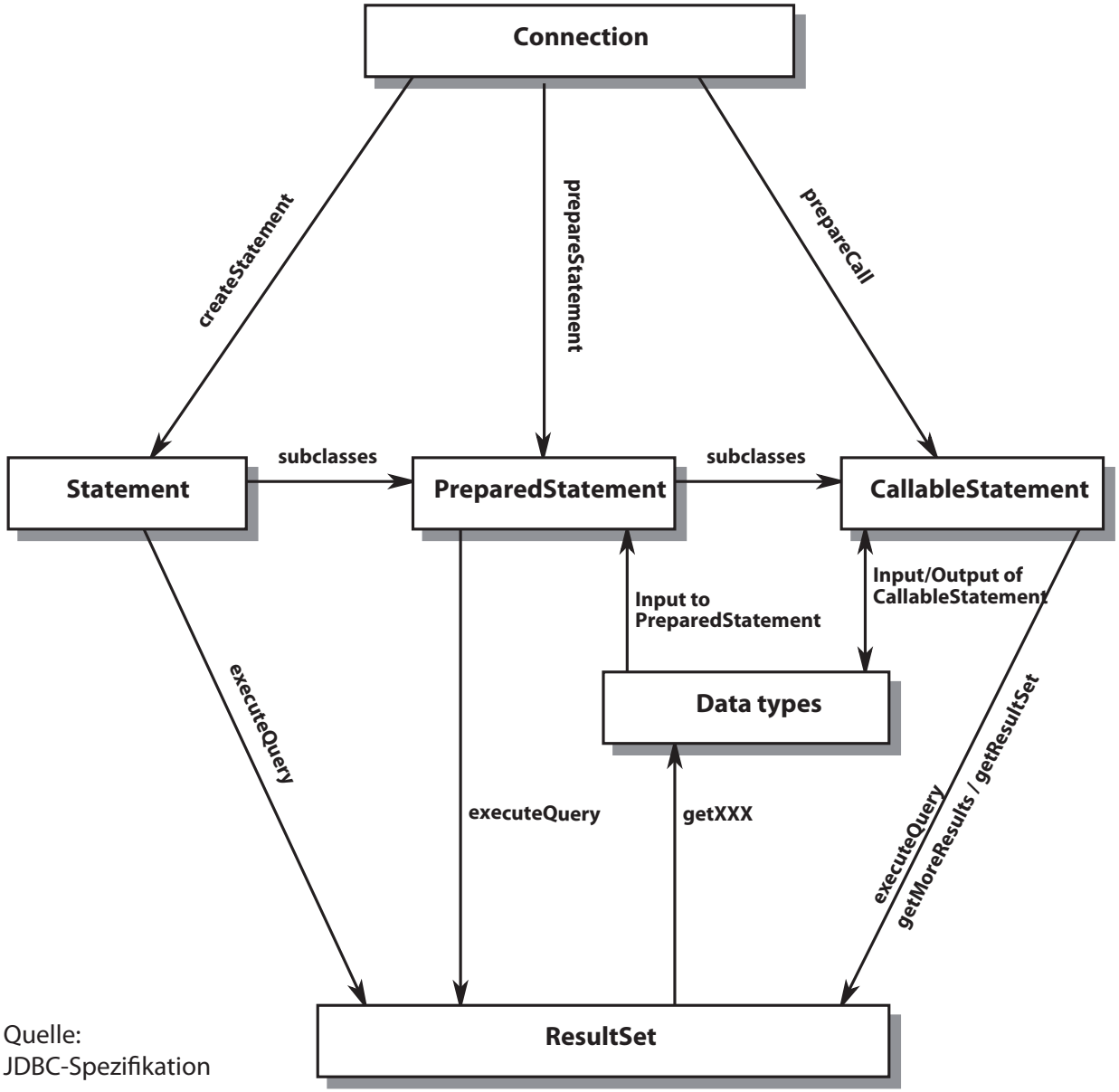
Architektur von JDBC



Typen von JDBC-Treiber



Interfaces und Klassen von JDBC



Quelle:
JDBC-Spezifikation

Connection

- Connection herstellen
  - `DriverManager.getConnection(url, user, passwd)`
  - `DataSource`
- URL einer Datenquelle
  - PostgreSQL

    `jdbc:postgresql:<Database>`

    `jdbc:postgresql://<Host>/<Database>`

    `jdbc:postgresql://<Host>:<Port>/<Database>`
  - MariaDB

    `jdbc:postgresql://<Host>:<Port>/<Database>?`
    `user=<uname>&passwd=pwd`

Statement

- `executeQuery`
- `executeUpdate`
- `execute`
- `executeBatch`

Parametrisierte Anweisungen

```
PreparedStatement pstmt = con.prepareStatement(
        "select author, title from Books where isbn = ?" );
pstmt.setString( 1, "0-201-70928-7" );
pstmt.executeQuery();

...

pstmt.setString( 1, "3-540-44008-9" );
pstmt.executeQuery();

...
```

Arten von ResultSets

- Art der Bewegung des Cursors
  - `TYPE_FORWARD_ONLY`
  - `TYPE_SCROLL_INSENSITIVE`
  - `TYPE_SCROLL_SENSITIVE`
- Lesender oder ändernder Cursor
  - `CONCUR_READ_ONLY`
  - `CONCUR_UPDATABLE`

Verwenden von ResultSets

- Navigieren
  - `next()`
  - `previous()`
  - `first()`
  - `last()`
  - `beforeFirst()`
  - `afterLast()`
  - `relative(int rows)`
  - `absolute(int r)`
- Werte lesen
  - `rs.getString(1)`
  - `rs.getString("author")`

Literatur zu JDBC

- Lance Andersen: *JDBC 4.2 Specification*, Oracle Inc., 2014
  `http://download.oracle.com/otndocs/jcp/jdbc-4_2-mrel2-spec/index.html`
- Burkhardt Renz: *JDBC - Kurze Einführung*, Vorlesungsskript, THM 2017
  `https://esb-dev.github.io/mat/JDBCIntro.pdf`

Übersicht

- Konzepte des Datenbankzugriffs
- JDBC
- ADO.NET
  - Grundlegendes Beispiel
  - Architektur von ADO.NET
  - Speicherresidente Datenbankstrukturen mit ADO.NET
- Objekt-relationales Mapping mit JPA

Grundlegendes Beispiel 1

```
using System;
using System.Data;
using System.Data.Odbc;

public class BooksReader {

  public static void Main() {

    /** Schritt 1: Connection zum Datenbanksystem vorbereiten */
    OdbcConnection con = new OdbcConnection(
      "DSN=azamon; UID=dis; PWD=ChrisDate");
```

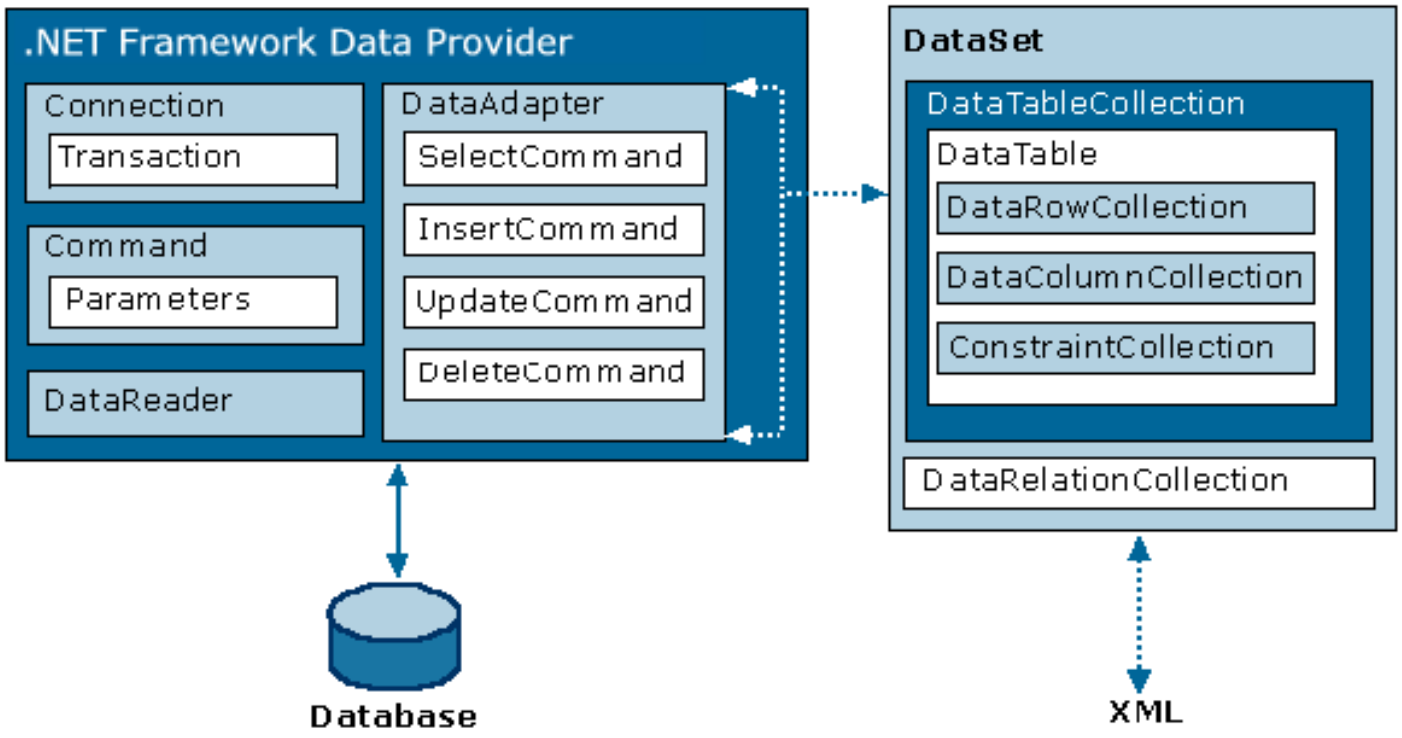
Grundlegendes Beispiel 2

```
/** Schritt 2: SQL-Kommando vorbereiten */
OdbcCommand cmd = new OdbcCommand(
   "select author, title from Books", con);

/** Schritt 3: Reader vorbereiten */
OdbcDataReader reader = null;

try {
  /** Schritt 3: Connection herstellen */
  con.Open();

  /** Schritt 4: SQL-Kommando ausfuehren */
  reader = cmd.ExecuteReader();

  /** Schritt 5: Ergebnis verwenden */
  while (reader.Read()) {
     Console.WriteLine(reader["author"].ToString()
       + " " + reader["title"].ToString());
 }
```

Grundlegendes Beispiel 3

```
    } catch (Exception e) {
      Console.WriteLine(e.ToString());
    }

    /** Schritt 6: Ressourcen freigeben */
    finally {
      if (reader != null ) reader.Close();
      if (con.State == ConnectionState.Open) con.Close();
    }
  }
}
```

Übersicht Klassen in ADO.NET



Data Provider

- Microsoft SQL Server
- OLE DB
- ODBC
- Oracle
- Oracle Data Provider ODP.NET von Oracle

Diskussion

Spezifische Klassen pro Provider
Provider Factory in ADO.NET 2.0

Wichtige Klassen

- Connection
- Command
- DataReader
- DataAdapter

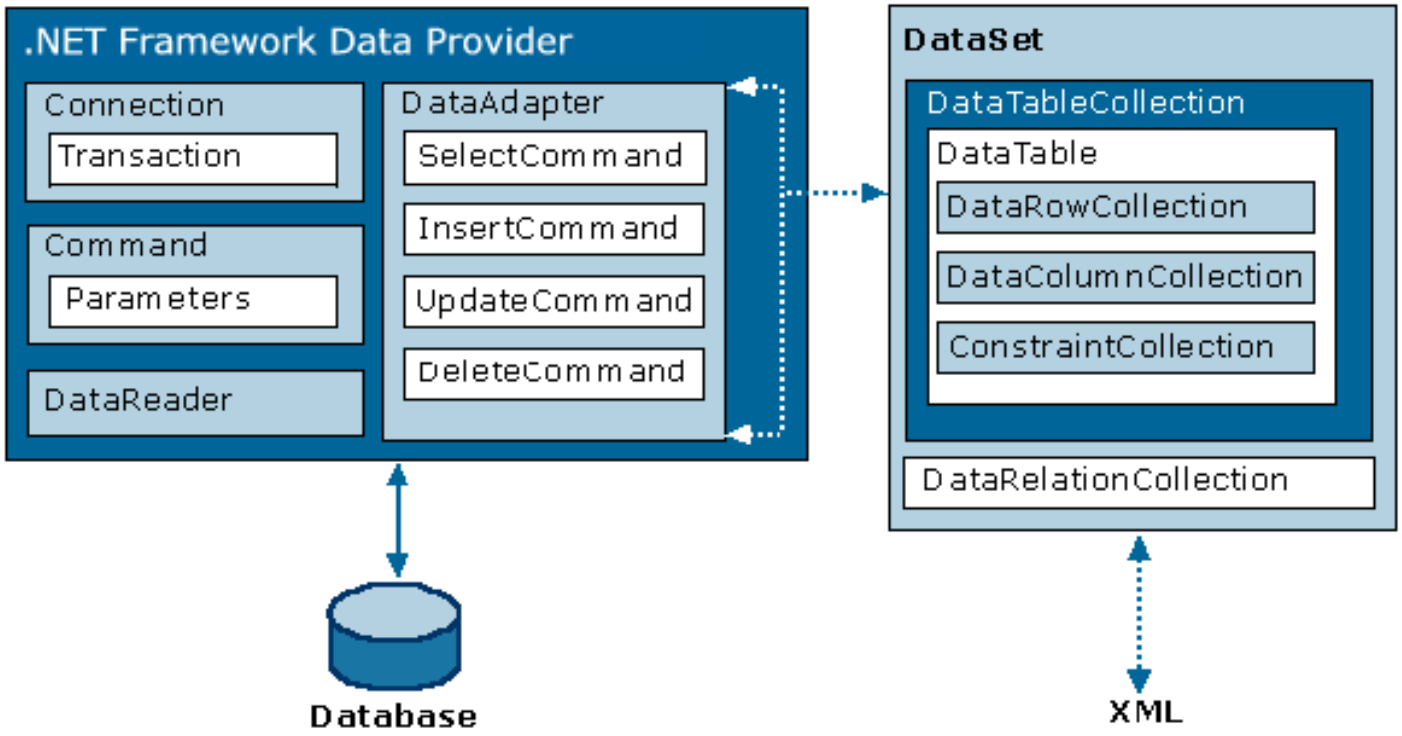
Arten der Ausführung einer Anweisung

- `ExecuteReader`
  → `DataReader`
- `ExecuteScalar`
  → Zahl
- `ExecuteNonQuery`
- Parametrisierte Anweisungen
- Stored Procedures

Metadaten

- Fehlermeldungen
  durch `DataException`
  Infos können an einen MessageHandler gebunden werden
- Indikatoren
  durch `DataReader::IsDBNUll(int i)`
- Aufbau der Ergebnismenge
  durch `DataReader::GetSchemaTable` → `DataTable`

Speicherresidente Datenbankstrukturen in ADO.NET



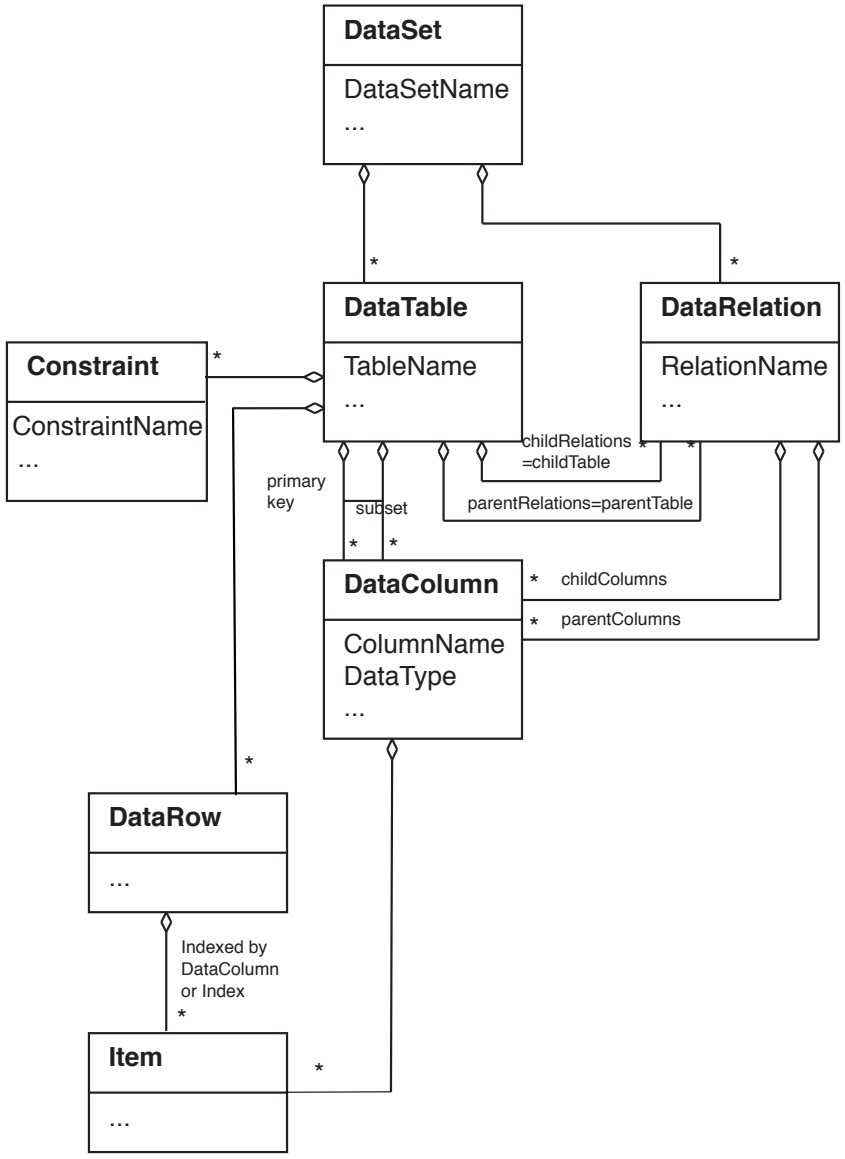
Wir betrachen nun den rechten Teil der Abbildung. Dazu:

Objektmodell einer relationalen Datenbank

Wie kann man eine relationale Datenbank als objekt-orientiertes Modell darstellen?

Diskussion

Aufbau von DataSet



DataSet

Konzept

Speicherresidente Datenbank („In-memory database“)

Klassen

- `DataTable`
- `DataRelation`
- `Constraints`
- `DataView`

DataAdapter

Konzept

Anbindung an Datenbank – Synchronisierung

Klassen

- `SelectCommand`
- `InsertCommand`
- `UpdateCommand`
- `DeleteCommand`
- `CommandBuilder`

Methoden

- `DataAdapter::Fill`
- `DataAdapter::Update`

Anbindung eines DataSets an GUI

- `DataGrid` dient zum Anzeigen von Daten
- viele Properties, die die *Optik* beeinflussen; auch `DataGridTableStyle`
- eine Menge Events, die gefeuert werden, wenn im Grid etwas passiert, z.B. Cursorbewegung, Ändern der Daten etc.
  etwa Event `RowChanging`
- `SetDataBindung` verbindet das `DataGrid` mit `DataTable`, `DataView` oder komplettes `DataSet` – gesteuert über `DataViewManager`

Übersicht

- Konzepte des Datenbankzugriffs
- JDBC
- ADO.NET
- Objekt-relationales Mapping mit JPA
  - Idee des ORM
  - Mapping von OO-Konzepten auf SQL-Datenstrukturen
  - Programmieren mit JPA

Idee des ORM

Konzeptbruch

- Werte vs. Referenztypen
- Primärschlüssel vs. Objektidentität
- Fremdschlüssel vs. Assoziationen
- Subclassing/Teilmengenbildung vs. Vererbung/Substitution

Idee

- Klassen, die Entitätstypen sind
- Objekt solcher Klassen sollen *per se* über die Fähigkeit der Persistenz verfügen

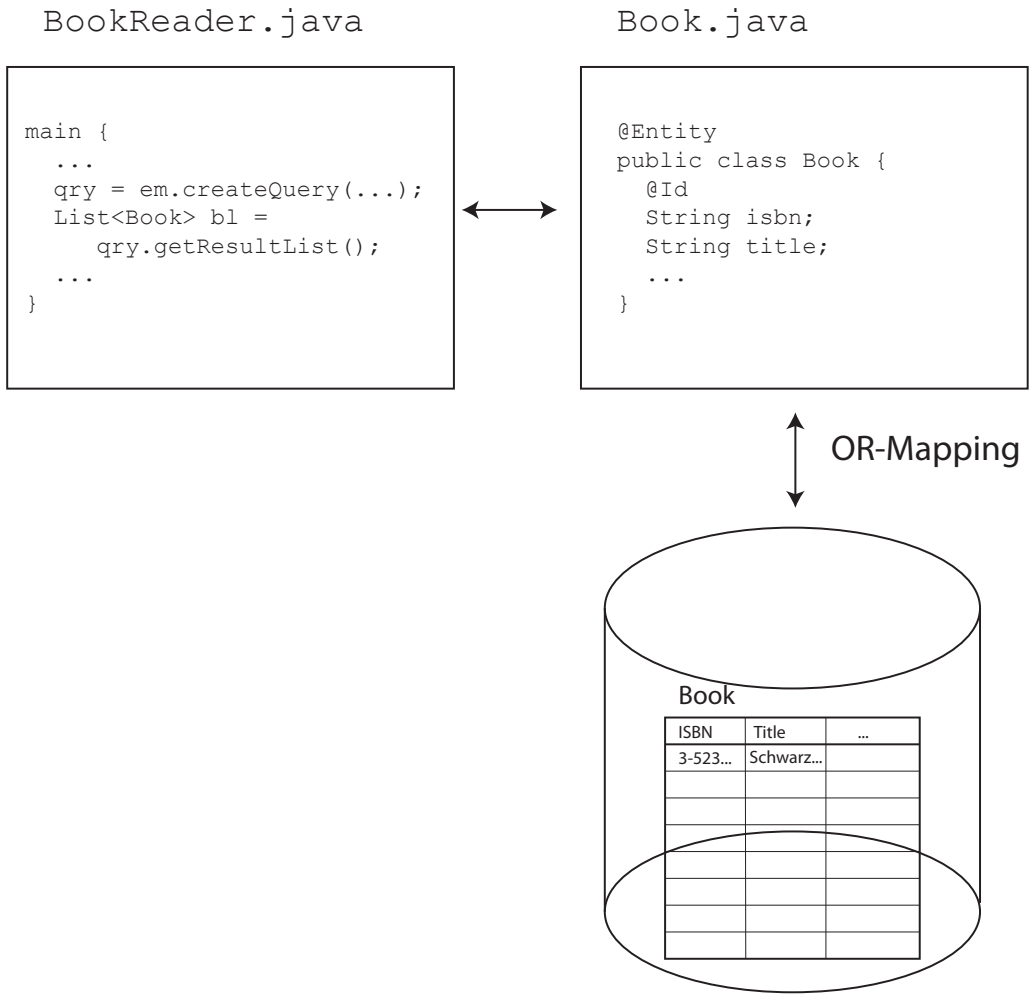
Grundlegendes Beispiel 1

```
import java.util.List;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.Query;

public class BookReader {

  public static void main( String[] args ) {

    // create EntityManager
    EntityManagerFactory emf =
      Persistence.createEntityManagerFactory( "azamon" );
    EntityManager em = emf.createEntityManager();
```

Grundlegendes Beispiel 2

```
    // create Query
    Query qry = em.createQuery( "select b from Book b" );

    // retrieve result
    List<Book> bookList = qry.getResultList();

    // print result
    for ( Book b: bookList ) {
      System.out.println( b.toString() );
    }

    // free resources
    em.close();
    emf.close();
  }
}
```

Konzept von JPA



Konzept von JPA

Was braucht man dazu?

- Zuordnung Klasse (Entitätstyp) – Tabelle
  → Annotationen in Java
- Maschine zur Verwaltung der Korrespondenz zur Laufzeit
  → EntityManager
- Zuordnung der Datenquelle
  → Persistence Unit in `persistence.xml`

persistence.xml

```
<persistence version="2.1"
             xmlns="http://xmlns.jcp.org/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
             http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
  <persistence-unit name="azamon" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>jpa.Book</class>
    <properties>
      <property name="javax.persistence.jdbc.user" value="dis"/>
      <property name="javax.persistence.jdbc.password" value="..."/>
      <property name="javax.persistence.jdbc.url"
      value="jdbc:postgresql://localhost/azamon"/>
      <property name="javax.persistence.jdbc.driver"
        value="org.postgresql.Driver"/>
    </properties>
  </persistence-unit>
</persistence>
```

Abbildung von Klassen und Assoziationen

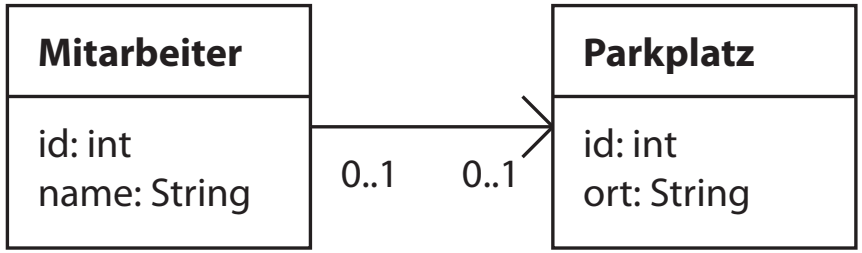
Ansatz

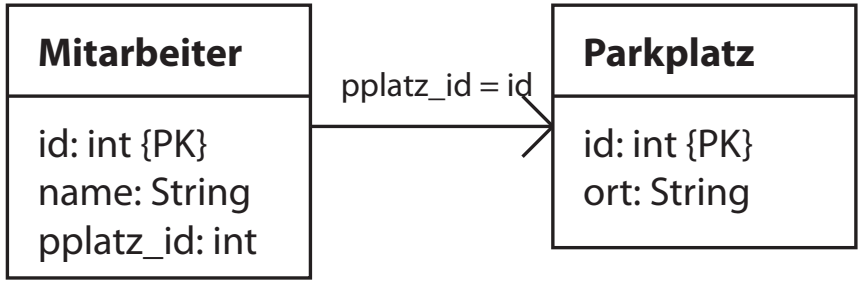
- Ausgangspunkt bestehende Datenbankstruktur
- Ausgangspunkt „Objektmodell“

Abbildung

- Klasse/Entitätstyp $\leftrightarrow$ Tabelle
- Assoziationen $\leftrightarrow$ Fremdschlüsselbeziehungen

One-to-One unidirektional



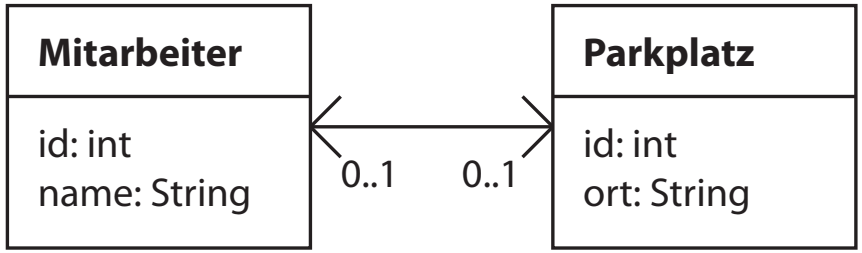


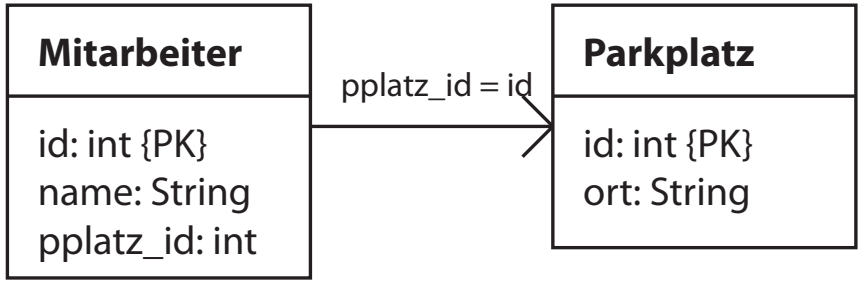
One-to-One unidirektional

```
@Entity
public class Mitarbeiter {
  @Id private int id;
  private String name;
  @OneToOne
  @JoinColumn(name="pplatz_id")
  private Parkplatz pplatz;
  ...
}
```

```
@Entity
public class Parkplatz {
  @Id private int id;
  private String ort;
  ...
}
```

One-to-One bidirektional



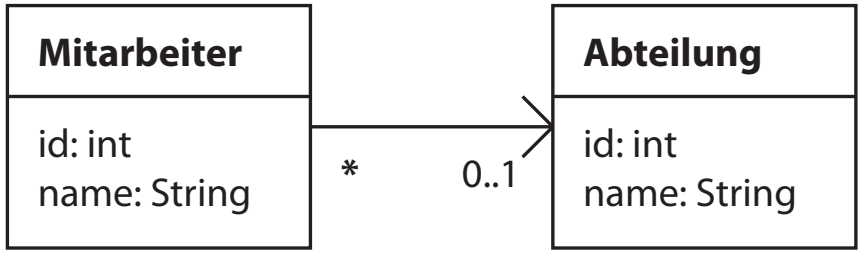


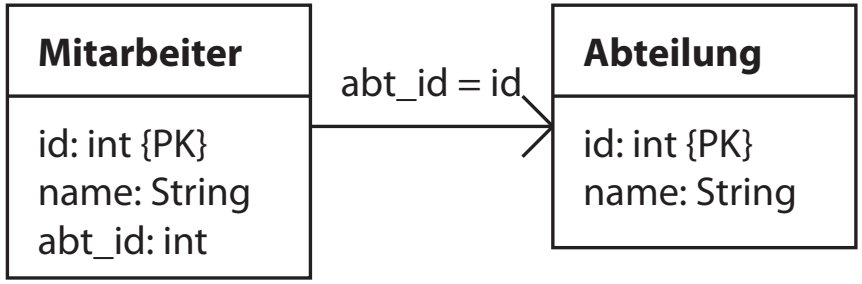
One-to-One bidirektional

```
@Entity
public class Mitarbeiter {
  @Id private int id;
  private String name;
  @OneToOne
  @JoinColumn(name="pplatz_id")
  private Parkplatz pplatz;
  ...
}
```

```
@Entity
public class Parkplatz {
  @Id private int id;
  private String ort;
  @OneToOne(mappedBy="pplatz")
  private Mitarbeiter mitarbeiter;
  ...
}
```

Many-to-One


Mitarbeiter
id: int
name: String
*
0..1
Abteilung
id: int
name: String

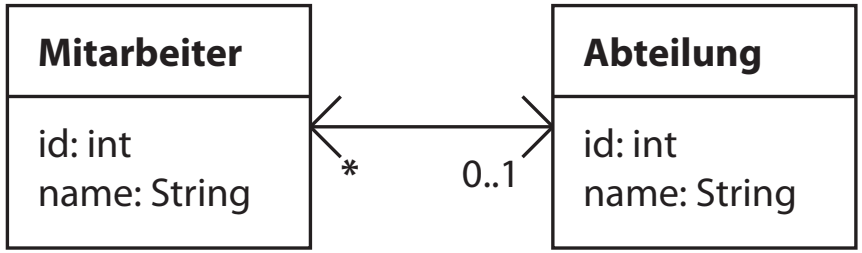

Mitarbeiter
id: int {PK}
name: String
abt_id: int
abt_id = id
Abteilung
id: int {PK}
name: String

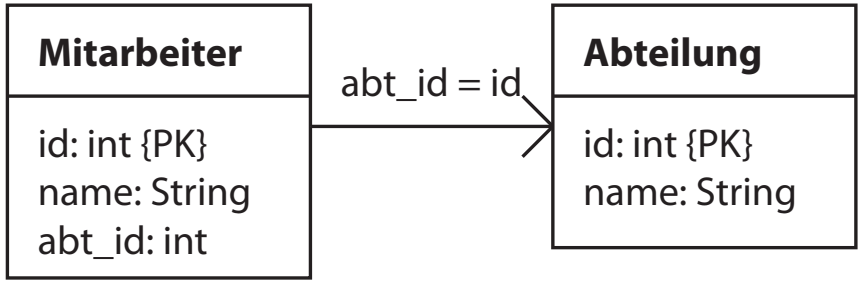
Many-to-One

```
@Entity
public class Mitarbeiter {
  @Id private int id;
  private String name;
  @ManyToOne
  @JoinColumn(name="abt_id")
  private Abteilung abt;
  ...
}
```

```
@Entity
public class Abteilung {
  @Id private int id;
  private String name;
  ...
}
```

One-to-Many / Many-to-One bidirektional



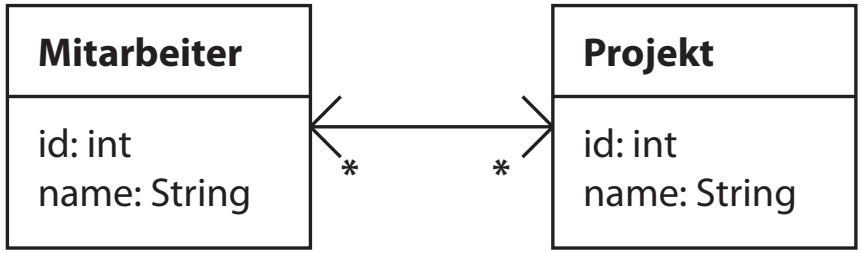


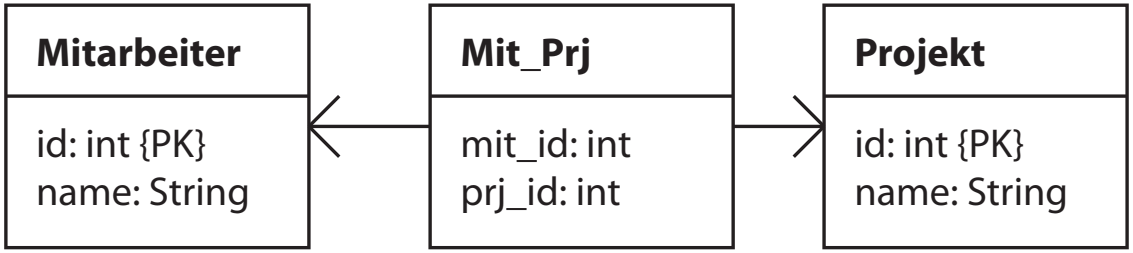
One-to-Many

```
@Entity
public class Mitarbeiter {
  @Id private int id;
  private String name;
  @ManyToOne
  @JoinColumn(name="abt_id")
  private Abteilung abt;
  ...
}
```

```
@Entity
public class Abteilung {
  @Id private int id;
  private String name;
  @OneToMany(mappedBy="abt")
  private Collection<Mitarbeiter>
      mitarbeiter;
  ...
}
```

Many-to-Many


Mitarbeiter
id: int
name: String
*
*
Projekt
id: int
name: String

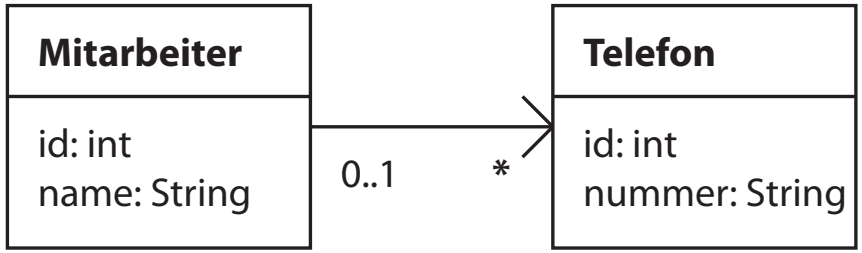

Mitarbeiter
id: int {PK}
name: String
Mit_Prj
mit_id: int
prj_id: int
Projekt
id: int {PK}
name: String

Many-to-Many

```
@Entity
public class Mitarbeiter {
  @Id private int id;
  private String name;
  @ManyToMany  [***]
  private Collection<Projekt>
     projekte;
  ...
}
[***]
@JoinTable(name="Mit_Prj",
    joinColumns=@JoinColumn(name="mit_id"),
    inverseJoinColumns=@JoinColumn(name="prj_id"))
```

```
@Entity
public class Projekt {
  @Id private int id;
  private String name;
  @ManyToMany(mappedBy="projekte")
  private Collection<Mitarbeiter>
      mitarbeiter;
  ...
}
```

One-to-Many unidirektional



One-to-Many unidirektional

```
@Entity
public class Mitarbeiter {
  @Id private int id;
  private String name;
  @OneToMany  [***]
  private Collection<Telefon>
     telefone;
  ...
}
[***]
@JoinTable(name="Mit_Tel",
    joinColumns=@JoinColumn(name="mit_id"),
    inverseJoinColumns=@JoinColumn(name="tel_id"))
```

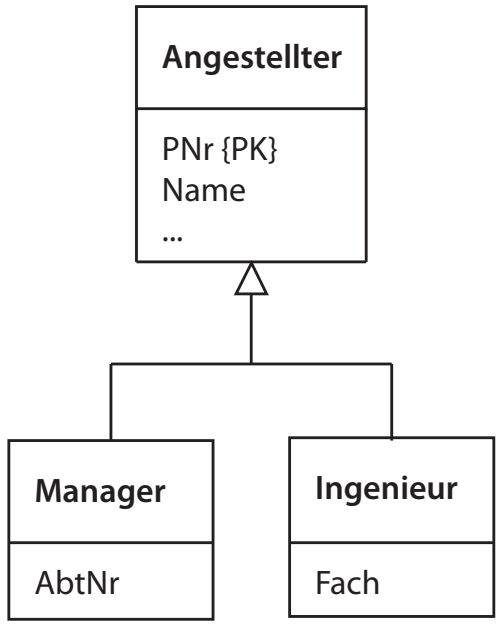
```
@Entity
public class Telefon {
  @Id private int id;
  private String nummer;
  ...
}
```

Strategien für das Mapping von Vererbung

- Eine Tabelle für die gesamte Klassenhierarchie
  `InheritanceType.SINGLE_TABLE`
- Eine Tabelle pro konkrete Entitätsklasse
  `InheritanceType.TABLE_PER_CLASS`
- Eine Tabelle pro Klasse
  `InheritanceType.JOINED`

Beispiel für die drei Strategien

Beispiel



Variante 1: Eine Tabelle für die komplette Hierarchie

`InheritanceType.SINGLE_TABLE`

```
create table Angestellter(
  pnr bigint primary key,
  dtype varchar(31),
  name varchar(255),
  abtnr integer,
  fach varchar(255)
);
```

Variante 2: Eine Tabelle pro konkreter Klasse

`InheritanceType.TABLE_PER_CLASS`

```
create table Angestellter(
  pnr bigint primary key,
  name varchar(255)
);
create table Manager(
  pnr bigint primary key,
  name varchar(255),
  abtnr integer
);
create table Ingenieur(
  pnr bigint primary key,
  name varchar(255),
  fach varchar(255)
);
```

Variante 3: Eine Tabelle pro Klasse

`InheritanceType.JOINED`

```
create table Angestellter(
  pnr bigint primary key,
  dtype varchar(31),
  name varchar(255)
);
create table Manager(
  pnr bigint primary key references Angestellter(pnr),
  abtnr integer
);
create table Ingenieur(
  pnr bigint primary key references Angestellter(pnr),
  fach varchar(255)
);
```

Einsatz von JPA

- EntityManager im Kontext eines Applikationsservers
  Enterprise Java Beans EJB
- EntityManager gesteuert durch eine Anwendung
  JPA in Java SE oder
  „Application-managed“ EntityManager in JavaEE

Persistenz-Kontext

- *Persistence Unit*: eine bestimmte Konfiguration von Entitätsklassen – Datenbank aus Sicht der Anwendung
- *Persistence Context*: die Menge der Objekte von Entitätsklassen, die der EntityManager steuert
- *Persistent Identity*: Identifikation eines persistenten Objekts – Primärschlüssel

Lebenszyklus eines Entitätsobjekts

- *New*: neu erzeugt, noch nicht einem Persistenz-Kontext zugeordnet
- *Managed*: hat eine persistente Identität und wird in einem Persistenz-Kontext verwaltet
- *Detached*: hat eine persistente Identität, wird aber zur Zeit nicht in einem Persistenz-Kontext verwaltet
- *Removed*: hat eine persistente Identität, ist verwaltet und muss bei der nächsten Synchronisierung mit der Datenbank dort gelöscht werden.

Wichtige Methoden

Methoden für Entitäten

- `find`: erzeugt Objekt aus der Datenbank

```
Mitarbeiter m = em.find(Mitarbeiter.class, id);
```

- `persist`: neue Objekte kommen in den Persistenz-Kontext

```
Mitarbeiter m = new Mitarbeiter(...);
em.persist(m);
```

- `remove`: markiert Objekte als zu löschend

```
Mitarbeiter m = em.find(Mitarbeiter.class, id);
em.remove(m);
```

Synchronisation mit der Datenbank

Synchronisation mit der Datenbank

- Aktionen werden kaskadierend durchgeführt
- Commit einer Transaktion
- expliziter Aufruf von `flush`

Strategien der Synchronisation

- Optimistische Strategie
  gesteuert über einen Timestamp, Annotation `@Version`
- Pessimistische Strategie
  durch explizite Sperren via EntityManager, Methode `lock`

JPQL Java Persistence Query Language

- Sprache bezieht sich auf das Objektmodell, *nicht* auf das Datenmodell des DBMS
- hat ähnlichen Aufbau wie SQL
- hat navigierende Syntax bei Assoziationen
- kann auch direkt SQL verwenden
- ...

Einfache Abfragen mit JPQL

```
Query q = em.createQuery( "select b from Buch b" );
List<Buch> rl = (List<Buch>) q.getResultList();

Query q = em.createQuery( "select b from Buch b where b.titel like 'A%'" );
List<Buch> rl = (List<Buch>) q.getResultList();

Query q = em.createQuery( "select b.titel from Buch b" );
List<String> rl = (List<String>) q.getResultList();
```

Spezielle Return-Werte

```
Query q.em.createQuery( "select b.titel, b.jahr from Buch b" );
List rl = q.getResultList();
for (Iterator i = rl.iterator(); i.hasNext();) {
  Object[] values = (Object[]) i.next();
  ...
}
```

```
// Definition von Klasse BuchKurz mit Titel und Jahr
...
Query q = em.createQuery( "select new BuchKurz(b.titel, b.jahr) from Buch b" );
List<BuchKurz> rl = (List<BuchKurz>) q.getResultList();
...
```

Assoziationen verwenden

Gegeben `Buch` mit einer Many-To-One-Assoziation zu `Verlag`

```
// Alle Buecher vom Springer-Verlag
... select b from Buch b where b.verlag.name = 'Springer' ...

// Alle Bueher vom Springer-verlag oder ohne Verlagsangabe
... select b from Buch b where b.verlag.name = 'Springer'
                            or b.verlag is null ....

// Alle Verlage, die ein Buch mit 'SQL' im Titel verlegt haben
... select v from Verlag v, in(v.buch) b where b.titel like '%SQL% ...
```

Fetch Joins

Gegeben wieder `Buch` mit einer Many-To-One-Assoziation zu `Verlag`.
Uns interessieren die Bücher zu einem Verlag

```
// Alle Buecher vom Springer-Verlag (prefetched)
... select v from Verlag v join fetch v.buch where v.name = 'Springer' ...
```

Benannte Abfragen (*named queries*)

```
// in der Klasse Buch
@NamedQuery(
  name="findBuchByTitel",
  queryString="select b from Buch b where b.titel = :titel"
)

// Verwendung
...
Query q = em.createNamedQuery( "findBuchByTitel" );
q.setParameter( "titel", "Schwarzrock" );
List<Buch> = (List<Buch>)q.getResultList();
...
```

Parametrisierte Abfragen

```
...
Query q = em.createQuery( "select b from Buch b
                    where titel like ?1 and jahr = ?2 ");
q.setParameter(1, "%SQL%");
q.setParameter(2, 2008);
...
```

SQL selbst verwenden

```
...
Query q = em.createNativeQuery(
    "select * from Buch where jahr between 1900 and 2000", Buch.class)
List<Buch> rl = q.getResultList();
...
```

JPA Criteria API

- JPA Query Language basiert (wie SQL) auf String-Repräsentationen der Anfragen
- In JPA 2.1 gibt es die Criteria API, in dem Anfragen als Objekt-Graph repräsentiert werden können.
- Setzt ein Metamodell der Datenquelle voraus

Beispiel

```
CriteriaBuilder cb = ...
CriteriaQuery<Customer> q = cb.createQuery(Customer.class);
Root<Customer> customer = q.from(Customer.class);
q.select(customer);
```

Entspricht „select c from Customer c“