Zusammenfassung

**Lernziele**

* Datenbanksysteme (DBMS) programmieren können (Stored Procedures, Triggers)
* Programmierschnittstellen für die Applikationsentwicklung vertiefter anwenden (JDBC)
* Werkzeuge zur DB-Anbindung an Applikationen einordnen können
* Grundlagen von objekt- relationalen sowie von objekt-orientierten DBMS kennen
* Semistrukturierte DBMS mit XML kennen
* Information Retrieval sowie Data Warehouse/OLAP einordnen können

**Unterlagen / Bücher**

* Elmasri & Navathe (2009): Fundamentals of Database Systems, Verlag AddisonWesley
* Ramakrishnan & Gehrke (2002): Data Base Management Systems, 3rd Ed., ISBN 0-07-115110-9
* Kemper & Eickler, Datenbanksysteme, Verlag Oldenbourg, ISBN 3-486-25706, 7. Auflage mit Übungsbuch
* Heuer & Saake (2000): Datenbanken: Konzepte und Sprachen, mitp-Verlag, 2. Auflage, 3-8266-0619-1

**Lerninhalte**

* Stored Procedures (PL/SQL)
* Cursors, Triggers, Udateable Views
* Backup und Recovery
* Indexe
* Objektrelationale Datenbanksysteme
* Objektorientierte Datenbanksysteme
* Semistrukturierte Daten (XML) in relationalen Datenbanksystemen
* Daten-Replikation und Synchronisierung
* In-Memory-Datenbanksysteme
* Datenstrukturen in Datenbanksystemen
* NoSQL-Datenbanksysteme
* Verteilte und Cloud Datenbanksysteme

Übersicht des Modules

[OR-Mapping (Einführung) 3](#_Toc475792556)

[Ideen und Lösungsansätze 3](#_Toc475792557)

[Semantic Gap in Persistence 3](#_Toc475792558)

[Object-Relation Mapper 3](#_Toc475792559)

[Technologien 3](#_Toc475792560)

[Custom Mappers 3](#_Toc475792561)

[Im Überblick 3](#_Toc475792562)

[Java Persistence API (JPA) 4](#_Toc475792563)

[Layers 4](#_Toc475792564)

[Provider Auswahl 4](#_Toc475792565)

[Grundgedanke 4](#_Toc475792566)

[JPA Kurz-Tour 4](#_Toc475792567)

[Mapping Regeln 5](#_Toc475792568)

[Regeln für Entites 5](#_Toc475792569)

[Table & Column Mapping 6](#_Toc475792570)

[Einfache Mapping Regeln 6](#_Toc475792571)

[Access Types 6](#_Toc475792572)

[Datentypen 6](#_Toc475792573)

[Mapping in XML 6](#_Toc475792574)

[Entity Relations (Beziehungen) 6](#_Toc475792575)

[1:1 Beziehung 6](#_Toc475792576)

[Bidirektionale 1:1 Beziehung 6](#_Toc475792577)

[Inverse 1:1 Beziehung 6](#_Toc475792578)

[N:1 Beziehung 6](#_Toc475792579)

[Bidirektionale N:1 Beziehung 6](#_Toc475792580)

[1:N Beziehung 6](#_Toc475792581)

[N:M Beziehung 6](#_Toc475792582)

[Bidirektionale N:M Beziehung 6](#_Toc475792583)

[Laden der Relations 6](#_Toc475792584)

[Ladestrategien 6](#_Toc475792585)

[«OR Mapping» mit JPA (Forsetzung) 7](#_Toc475792586)

[Wie kommen Entites und Daten zusammen? 7](#_Toc475792587)

[JPA-Architektur (Begriffe) 7](#_Toc475792588)

[Kontrolle über Verhalten und Zustände der Entities 7](#_Toc475792589)

[Entity-Identität 7](#_Toc475792590)

[Änderungen an Relations 7](#_Toc475792591)

[Transaktionen 7](#_Toc475792592)

[Vererbung 7](#_Toc475792593)

[Anfragen in JPA mit JPQL 7](#_Toc475792594)

# OR-Mapping (Einführung)

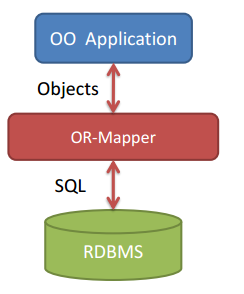
## Ideen und Lösungsansätze

In Software-Programmen möchte man Objekte manipulieren (CRUD), insbesondere speichern (=persistieren) und wieder holen (=abfragen).

Mit reinem JDBC muss ich jede Spalte auf Java Objekte ummappen. Ein OR Mapper soll nun das Objektmodell und das Datenbankmodell verbinden. Dabei möchte ich nichts mit dem Mapping vom Objektmodell zum Datenbankmodell zu tun haben.

## Semantic Gap in Persistence

OR Mapper sollen diesen Semantic Gap zwischen den beiden Modellen schliessen. Die Programme sind meist objekt-orientiert, während die Datenbanken meist relational sind.

Es soll eine «Nahtlose» Abbildung von und zu der darunterliegenden DB sein.

**Funktionen**

* Laden von relationalen Daten als Verbund von Objekten (Joins über mehrere Tabellen)
* Speichern von Änderungen an persistenten Objekten

## Object-Relation Mapper

### Technologien

**Unter Java** Java Persistence API (JPA), Java Data Objects (JDO), Hibernate

**Unter .NET** Entity Framework (EF), NHibternate

### Custom Mappers

In gewissen Fällen sinnvoll

* Bei besonders hoher Abbildungskomplexität (Mappings auf SPs)
* Bei mehreren Datensilos (SQL, NoSQL, Streams, Filesystems)
* Evtl. höhere Performance
* Evtl. besondere Skalierung
* Besondere Transaktionsverwaltung

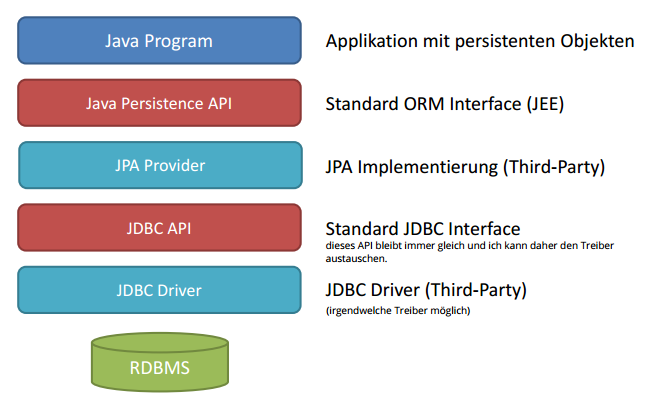
Custom Mappers haben natürlich einen hohen Arbeitsaufwand und die Komplexität ist auch vorhanden.

## Im Überblick

Das **Java Persistence API** (JPA) ist das Standard OR Mapping zu relationalen Datenbanken. Es ist Teil von Enterprise Java Beans, läuft aber auch ohne JEE und ohne JEE Application Server. Es ist nur eine Spezifikation. Diverse Implementierungen vorhanden. Beeinflusst wird es durch Hibernate und Java Data Object.  
**Java Data Object** (sozusagen tod), Standard vor JPA

# Java Persistence API (JPA)

## Layers



## Provider Auswahl

Es gibt verschiedene JPA Implementierungen. Dazu zählen EclipseLink, Hibernate, Apache OpenJPA, DataNucleues,….. In den Beispielen in dieser Vorlesung wird EclipseLink verwendet.

## Grundgedanke

**Entities** als persistente Objekte. Dabei sollen POJO («Plain Old Java Objekte») verwendet werden, also ganz normale Java Objekte. Mit einem ID-Feld für die Abbildung auf den Primarschlüssel in der Datenbank. Zudem ein Parameter losen Konstruktor.

Die **Abbildung** zur Datenbank wird mittels Annotations im Code (nicht ganz POJO) realisiert. Alternativ kann dies auch in separaten Mapping-Files deklariert werden.

Die **Anbindung** an die Datenbank findet im Persistence.XML Konfigurationsfile statt.

## JPA Kurz-Tour

|  |  |
| --- | --- |
| **Entity Klasse** | **Abbildung auf die DB** |
| **JPA Konfiguration** File «persistence.xml » im Folder « META-INF» |  |
| **Laden/Suchen von Entities** | **Explizites Einfügen von Entities** |
| **Update von Entities** | **Explizites Löschen von Entities**  Zum Löschen muss das Objekt zuerst in den Cache geladen werden und kann dann erst gelöscht werden. Mit commit wird dann die Löschoperation erst effektiv ausgeführt wird. |

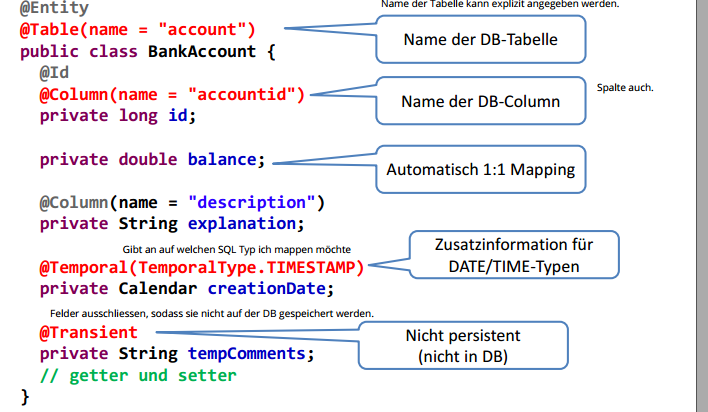
## Mapping Regeln

### Regeln für Entites

Eine Entity ist eine Klasse mit der Annotation @Entity. Diese Klasse kann erben, vererben, kann Interface implementieren oder auch abstrakt sein. Von dieser Seite her gibt es also keine Einschränkungen. Es gibt aber Einschränkungen bezogen auch den Inhalt der Klasse.

* Es muss ein Public oder Protected Konstruktor ohne Argumente vorhanden sein.
* Es ist eine Primary Key Angaben erforderlich (Annotation @id)
* Klasse darf nicht final sein, auch auch keine final Flieds oder Methoden haben
* Die Fileds sollen private oder protected sein und nur über getter/Setter zugreifbar.

### Table & Column Mapping

Mit @Table kann der Name der Tabelle explizit angegeben werden.

Mit einer Spalte ist dies über @Column möglich.

Mit @Temportal kann ich den entsprechenden SQL-Typ angeben.

Über @Transient kann ich Felder ausschliessen.

### Einfache Mapping Regeln

Standardmässig findet das Mapping mit den gleiche Namen statt. Für die Spalten und die Tabellen kann dies explizit angegeben werden. Es ist Case Insensitiv.

Per Default werden alle Attribute in das Mapping einbezogen. Ausnahmen sind die entsprechenden Annotationen.

Bei Calendar/Date Java-Typ braucht es Zusatzinformationen. Dafür lässt sich die Temporal Annotation verwenden. Eine Abbildung ist auch Date, Time oder Timestamp möglich. Aber aufpassen, die Unterstützung ist nicht immer gleich.

### Access Types

|  |  |
| --- | --- |
| **Field Access** DB-Attribute direkt in Fields abbilden | **Property Access** DB-Attribute über Getter/Setter abbilden |

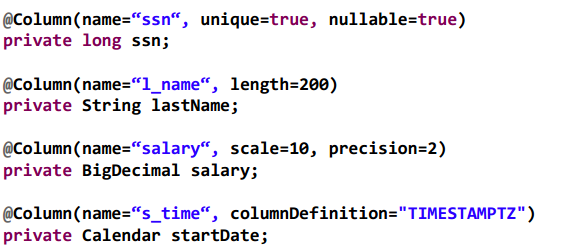
Entweder macht man Field oder Property Access pro Klasse. Die Attribut-Annotation bestimmt den Access Type. Mindestens @Id Annotation ist pro Klasse nötig.

Beim Propery Access die Attribute Annotation beim Getter angeben. Alle Properties müssen Getter und Setter haben. Statt getXYZ() ist auch isXYZ() möglich (passend für den Boolean)

### Datentypen

Es werden folgende **Typen** beim Mapping **unterstützt**:  
Primitive Typen und deren Wrapper-Typen, String, Enums, Byte und Character Array, Date, Calendar und beliebige Serializierbare Klassen (welche nach BLOB abgebildet werden).

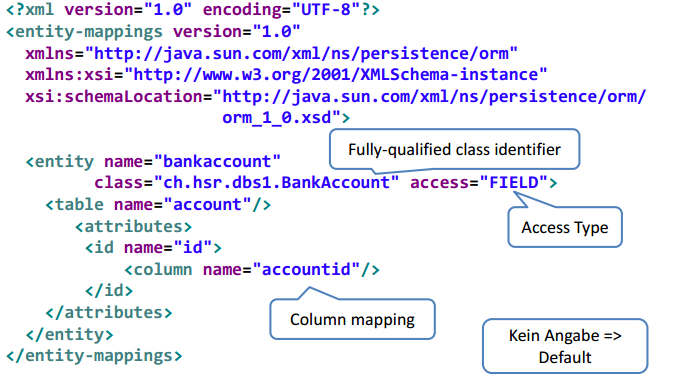
Zu den **unterstützten Relationen** zählen alle Referenzen auf Instanzen mit Entity Klassen. Darunter Collection<>, Set<>,List<> und Maps<> Entities.

Es ist auch eine explizite Detail-Angabe vom DB-Typ möglich. Nützlich für automatische Schema-Erzeugung von JPA.  


### Mapping in XML

Eine Alternative zu den Annotationen in den Java Klassen. Dazu findet die Abbildung in orm.xml im Folder META-INF statt. Die Prioritäten des Mappings sind wie folgt festgelegt.

1. XML-File orm.xml
2. Entity-Annotationen
3. Default-Mapping



## Entity Relations (Beziehungen)

Die Beziehungen können Unidirektional oder bidirektional sein. Es existieren verschiedene Kardinalitäten. In der DB als Tabellen mit FK-Schlüssel Beziehungen. Es ist eine Explizite Annotation oder XML-Mapping nötig.

### 1:1 Beziehung

|  |  |
| --- | --- |
|  |  |

### Bidirektionale 1:1 Beziehung

|  |  |
| --- | --- |
|  |  |

### Inverse 1:1 Beziehung

|  |  |
| --- | --- |
|  |  |

### N:1 Beziehung

|  |  |
| --- | --- |
|  |  |

### Bidirektionale N:1 Beziehung

|  |  |
| --- | --- |
|  |  |

### 1:N Beziehung

|  |  |
| --- | --- |
|  |  |

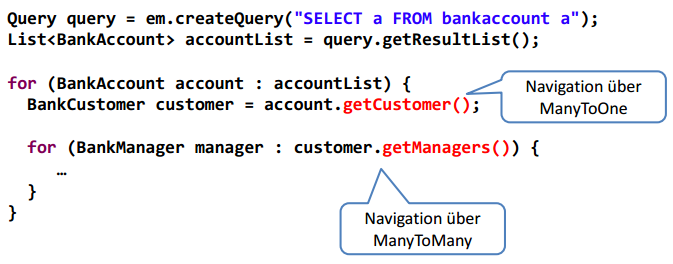
### N:M Beziehung

|  |  |
| --- | --- |
|  |  |

### Bidirektionale N:M Beziehung

|  |  |
| --- | --- |
|  |  |

### Laden der Relations

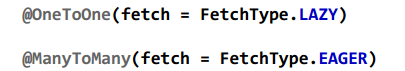
Die Normale Travesierung des Objektgraphen. Objekte laden, welche unter Abhängigkeit von einander stehen.  


### Ladestrategien

**Eager Loading**  
Target Entity wird direkt mit der Beziehung geladen. Default bei @OneToOne und @ManyToOne.

**Lazy Loading**  
Laden der Entity beim ersten Zugriff auf die Beziehung. Default bei @OneToMany und @ManyToMany.

Die Ladestrategien können explizit mit Annotationen gesteuert werden. Es ist ein bisschen doof, da man es bei jeder Beziehung wieder angeben muss. Es gibt andere Entities Frameworks, welche das besser machen.



# «OR Mapping» mit JPA (Forsetzung)

## Wie kommen Entites und Daten zusammen?

## JPA-Architektur (Begriffe)

## Kontrolle über Verhalten und Zustände der Entities

## Entity-Identität

## Änderungen an Relations

## Transaktionen

## Vererbung

## Anfragen in JPA mit JPQL