Datenbanken und Informationssysteme – 4. JG

JPA

Foliensatz

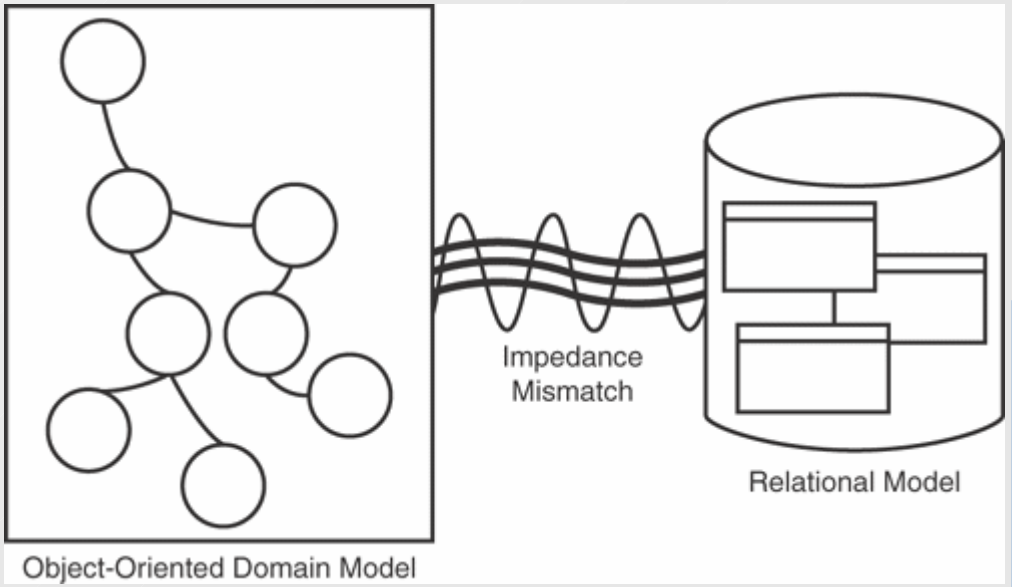
DI(FH) Gerald Aistleitner

Was ist JPA?

= **J**ava **P**ersistence **A**PI

Schnittstelle für Java-Anwendungen, die die Zuordnung und die Übertragung von Objekten zu Datenbankeinträgen vereinfacht.

Vereinfacht die Problematik der objektrelationalen Abbildung (OR-Mapping)



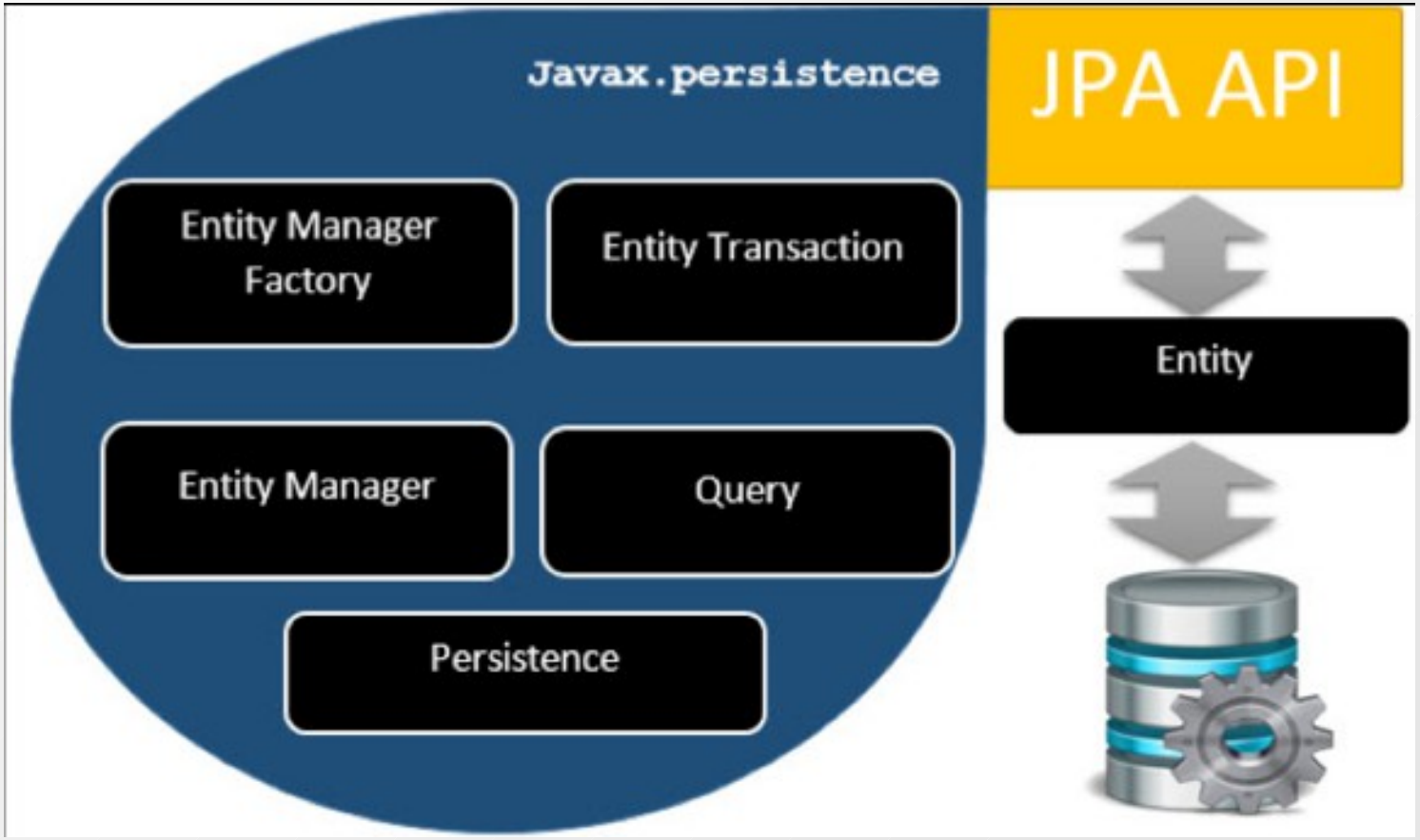
Geschichte / Provider

- Mit EJB 3.0 wurde der Persistence Layer ausgelagert in JPA 1.0
- JPA 2.0 mit Java EE6 (2009)
- JPA 2.1 mit Java EE7 (2013)
- JPA 2.2

Provider:

- Hibernate
- Eclipselink
- Toplink
- Spring Data JPA

JPA - Klassen



Quelle: tutorialspoint.com

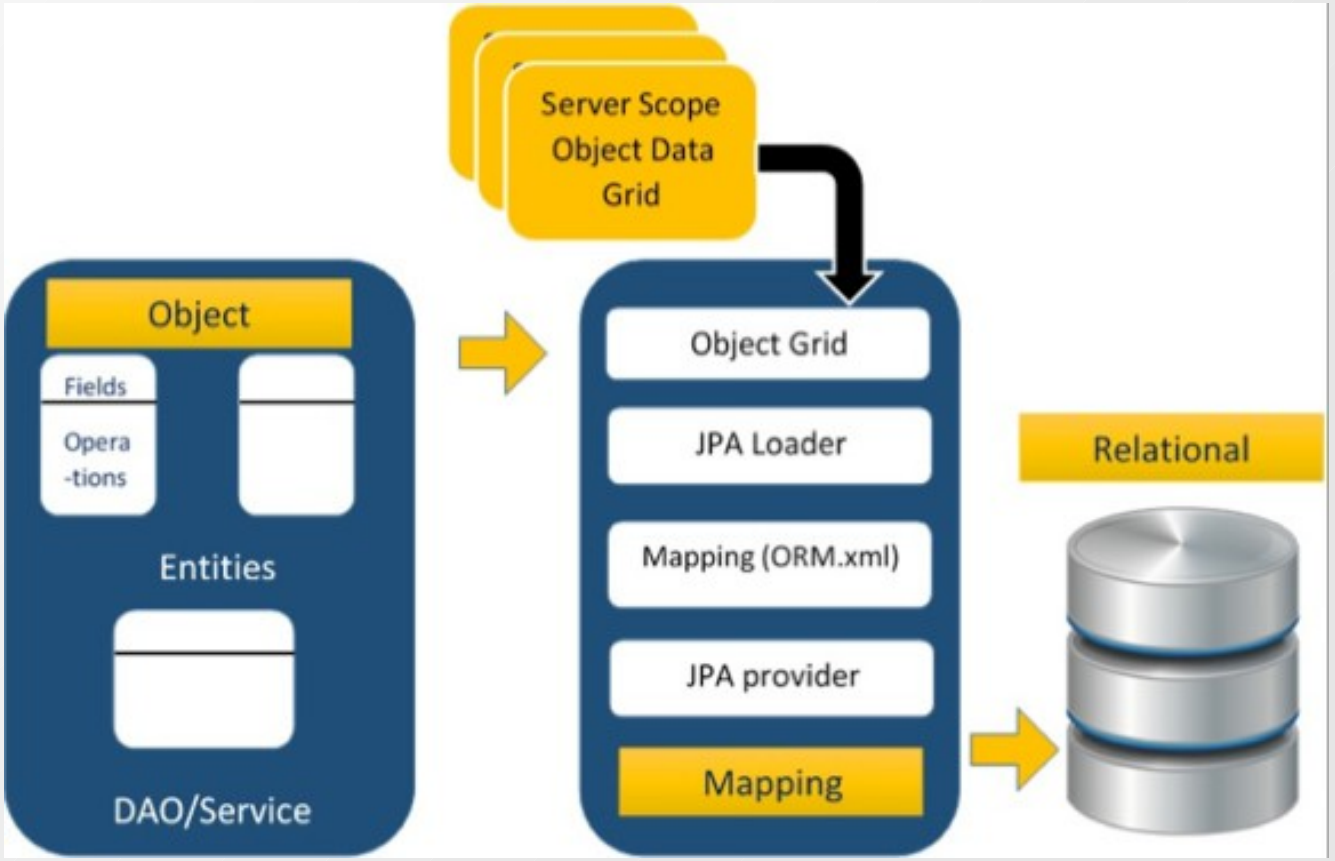
ORM – Object Relational Mapping

Ermöglicht die Konvertierung von Objekttypen zu relationalen Typen und umgekehrt.

Hauptaufgabe ist das Mapping/Binding von Objekten zu ihren Daten in der Datenbank.

ORM – 3 Phasen

- Phase 1: Object Data Phase
- Phase 2: Mapping/Persistence Phase
- Phase 3: Relational Data Phase



Quelle: tutorialspoint.com

Mapping.xml

```
<? xml version="1.0" encoding="UTF-8" ?>

<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm
   http://java.sun.com/xml/ns/persistence/orm_1_0.xsd"
   version="1.0">

   <description> XML Mapping file</description>

   <entity class="Employee">
      <table name="EMPLOYEETABLE"/>
      <attributes>

         <id name="eid">
            <generated-value strategy="TABLE"/>
         </id>

         <basic name="ename">
            <column name="EMP_NAME" length="100"/>
         </basic>

         <basic name="salary">
         </basic>

         <basic name="deg">
         </basic>

      </attributes>
   </entity>

</entity-mappings>
```

Persistable Types

- User defined classes
  Entity classes, Mapped Superclasses,
  Embeddable classes
- Simple Java data types
  Primitive types, Wrappers, String, Date, Math types
- Multi value types
  Collections, Maps, Arrays
- Miscellaneous types
  Enum, Serializable types

Entitäten

Um eine Klasse persistieren zu können, müssen folgende Punkte erfüllt sein:

- Annotation mit @Entity auf Klassenebene
- Annotation mit @Id für Primärschlüssel
- Default-Konstruktor
- Serializable implementieren

```
@Entity
public class Products implements Serializable {
    @Id
    private Integer productid;

    private String productname;
   …
}
```

Primärschlüssel

Jedes Entity-Objekt benötigt einen Primärschlüssel.
@Id

Automatische Generierung

- Table-Strategy

```
@Entity
@TableGenerator(name="idtab", initialValue=0, allocationSize=10)
public class Mitarbeiter implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.TABLE, generator="idtab")
    private Long id;
```

- Sequence-Strategy

```
@SequenceGenerator(name="maseq", initialValue=0, allocationSize=10)

  @Id
  @GeneratedValue(strategy = GenerationType.SEQUENCE, generator="maseq")
```

- Auto (Persistence Provider entscheidet)

Weitere Annotationen

- @Table: Benennung der Datenbanktabelle
  - name: Name der Tabelle
- @Column: Steuerung der DB-Column, Attribute:
  - name: Name der DB-Spalte
  - length: Länge des Datenfeldes
  - nullable: Darf NULL eingefügt werden?
  - precision, scale: Steuerung von num. Datentypen
- @Basic: erlaubt das setzen von Attributen wie
  - optional: setzt entsprechend der Wahl NOT NULL
  - fetch: eager oder lazy

Persistence Unit

Definiert in persistence.xml (META-INF)

```xml
<?xml version="1.0" encoding="UTF-8"?>
<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="JPA_01PU" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <class>jpa_01.Mitarbeiter</class>
    <properties>
      <property name="javax.persistence.jdbc.url" value="jdbc:derby:c:\temp\db01"/>
      <property name="javax.persistence.jdbc.user" value=""/>
      <property name="javax.persistence.jdbc.driver" value="org.apache.derby.jdbc.EmbeddedDriver"/>
      <property name="javax.persistence.jdbc.password" value=""/>
      <property name="hibernate.cache.provider_class" value="org.hibernate.cache.NoCacheProvider"/>
      <property name="javax.persistence.schema-generation.database.action" value="create"/>
    </properties>
  </persistence-unit>
</persistence>
```

Generierung der DB

```
<property name="javax.persistence.schema-generation.database.action"
value="drop-and-create"/>
```

none	Es werden keine Tabellen generiert. => Produktivsystem
create	Tabellen werden generiert. Existieren diese bereits, wird nichts geändert.
drop-and-create	Alle Objekte der DB werden gelöscht und vom Provider neu erstellt
drop	Alle Objekte in der DB werden beim Deployment gelöscht

Hibernate

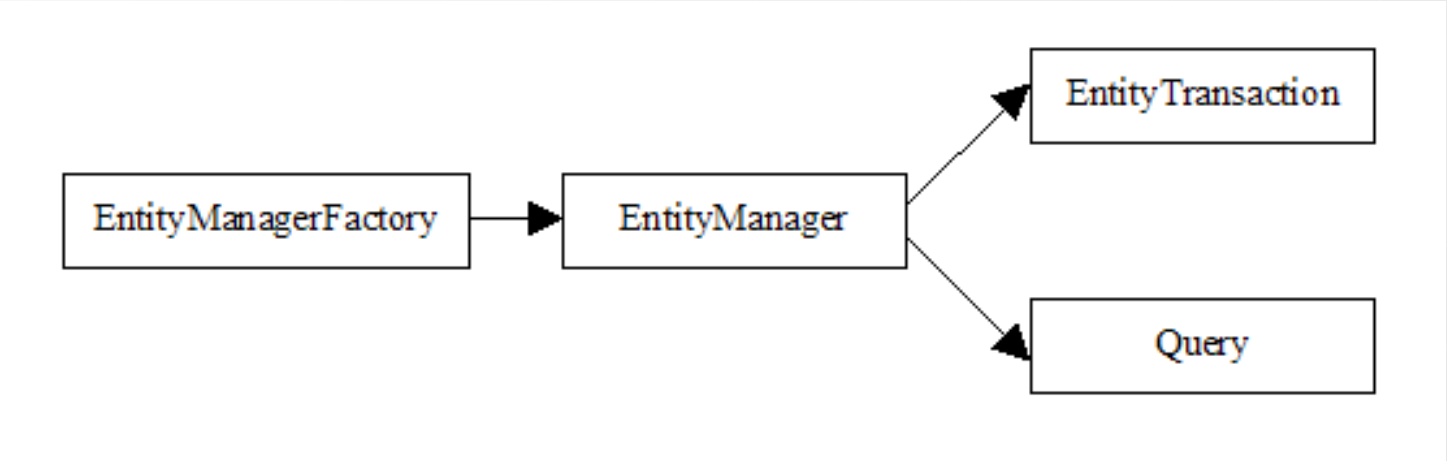
For `hbm2ddl.auto` property the list of possible options is:

- **validate**: validate that the schema matches, make no changes to the schema of the database, you probably want this for production.
- **update**: update the schema to reflect the entities being persisted
- **create**: creates the schema necessary for your entities, destroying any previous data.
- **create-drop**: create the schema as in **create** above, but also drop the schema at the end of the session. This is great in early development or for testing.

http://antoniogoncalves.org/2014/12/11/generating-database-schemas-with-jpa-2-1/
https://docs.oracle.com/javaee/7/tutorial/persistence-intro005.htm

Datenbankverbindung via JPA

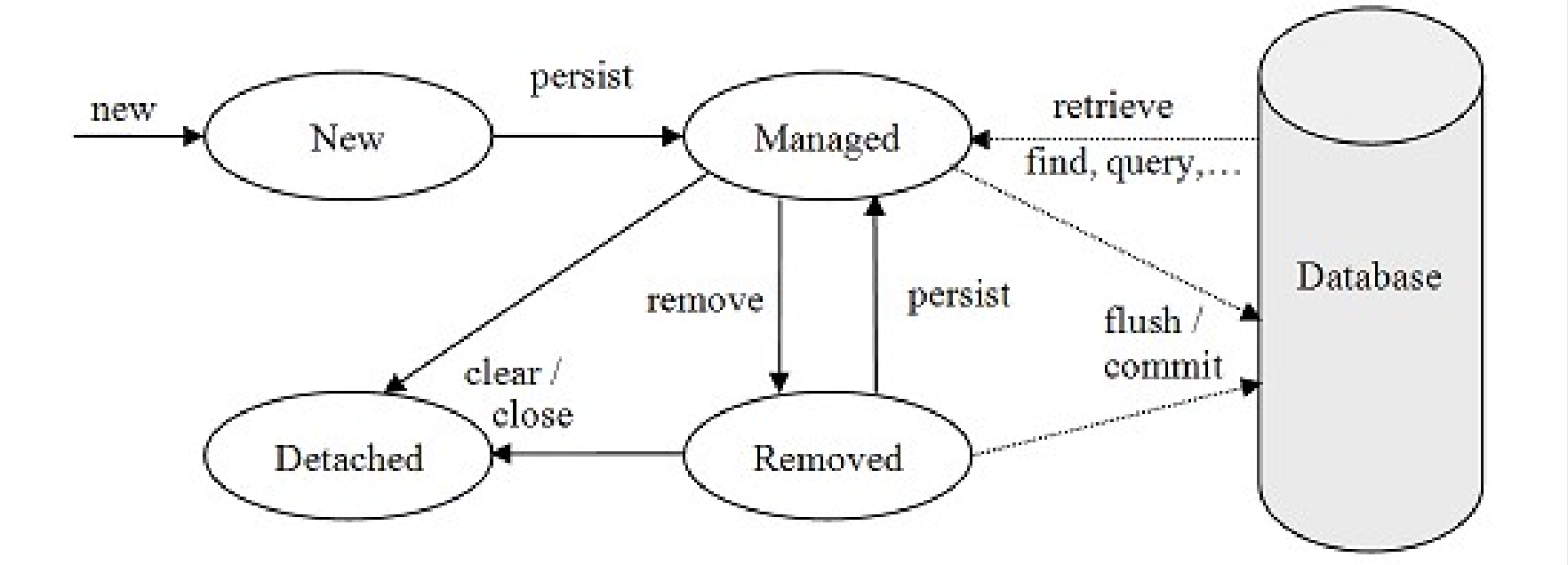
Für den Datenbankzugriff werden folgende Schnittstellen verwendet:



Quelle: objectdb.com

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("JPA_01PU");
EntityManager em = emf.createEntityManager();

em.getTransaction().begin();
// Manipulation Code
em.getTransaction().commit();
em.close();
emf.close();
```

LifeCycle von Entity-Objekten



Persistence Context

- Enthält alle Objekte des EntityManagers im Status Managed
- Wird ein Objekt abgerufen, das bereits im Persistence Context ist, wird nicht erneut auf die Datenbank zugegriffen, sondern direkt vom Persistence Context geliefert (Cache)

CRUD - Create

- Das neue Objekt wird dem EntityManager zum persistieren übergeben. Das ganze muss in einer Transaktion stattfinden (EE automatisch, SE händische Transaktionssteuerung):

```
Person person =
                new Person(1, "Max", "Muster");
em.persist(person);
```

CRUD - READ

- Das Objekt kann wiederum mittels EntityManager über den jeweiligen Primärschlüssel geladen werden:

```
Person person = em.find(Person.class, 1L);
```

- Alternativ können komplexere Abfragen über das QueryAPI ausgeführt werden (siehe später).

CRUD - UPDATE

- Objekte im managed-Zustand werden innerhalb einer Transaktion automatisch auch in der DB aktualisiert (commit).

  Alternativ kann auch die merge-Methode aufgerufen werden. Dies aktualisiert den Datensatz, führt das Objekt aber nicht in den Managed-Zustand über. D.h. Änderungen nach Aufruf von merge werden nicht in die Datenbank übernommen.

```
em.merge(person);
```

CRUD - DELETE

- Durch Aufruf von remove wird der Datensatz gelöscht (nach commit):.

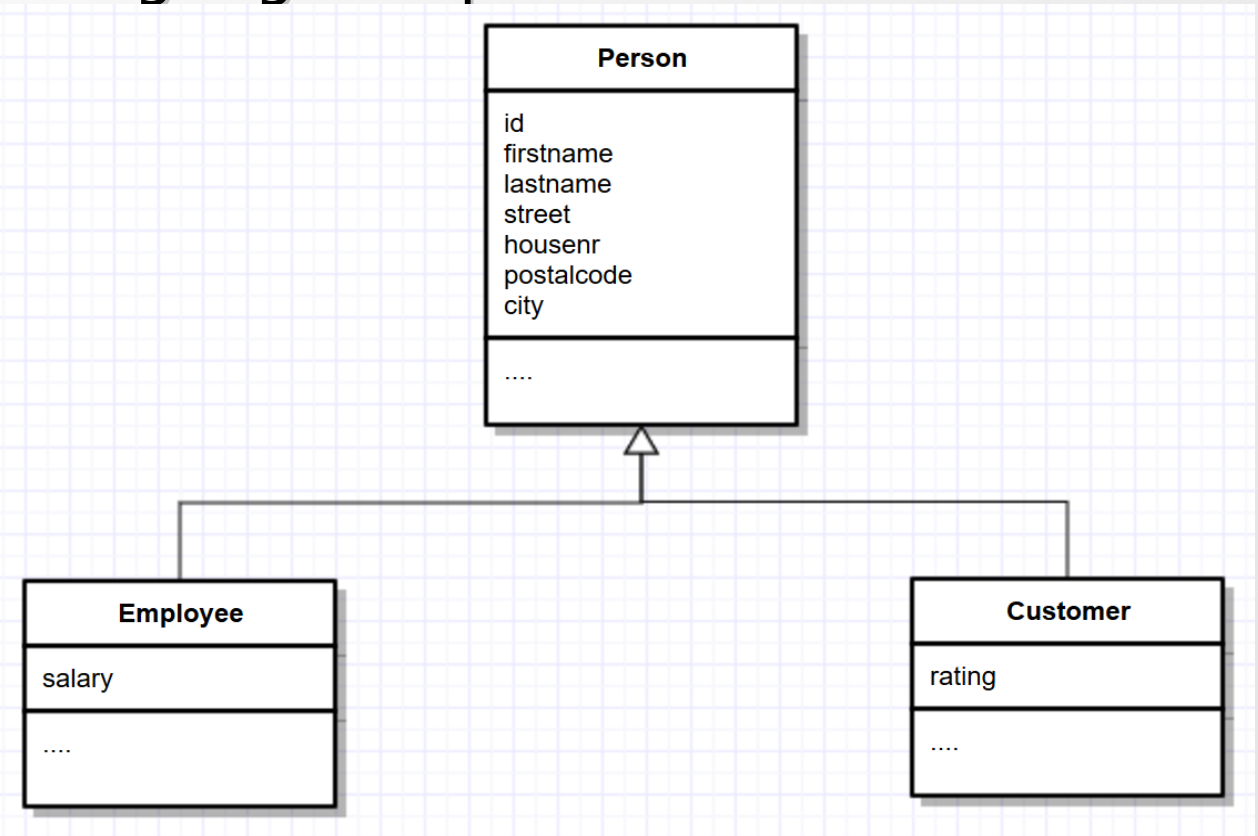
```
em.remove(person);
```

Vererbung und Beziehungen zw. Entitäten

Vererbung

Wie kann eine klassische Vererbungshierarchie in der Datenbank abgebildet werden?

Ausgangsbeispiel:




```
@Inheritance (
   strategy=...)
```

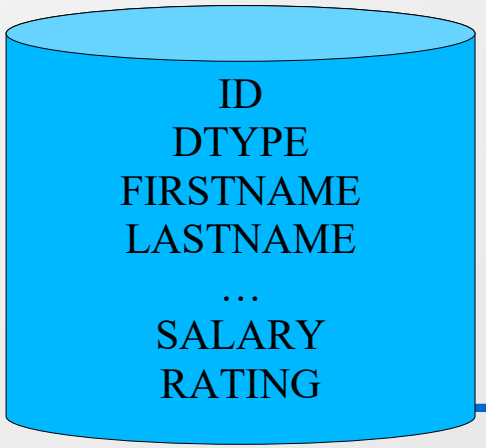
3 Strategien:

- JOINED
- SINGLE_TABLE
- TABLE_PER_CLASS

Vererbung - SINGLE_TABLE

Es wird eine Tabelle generiert, die **sämtliche Attribute** aller beteiligten Klassen enthält. Zusätzlich wird ein sogenannter **Diskriminator** (DTYPE) erstellt, der festlegt, welchen Typ der Datensatz darstellt.

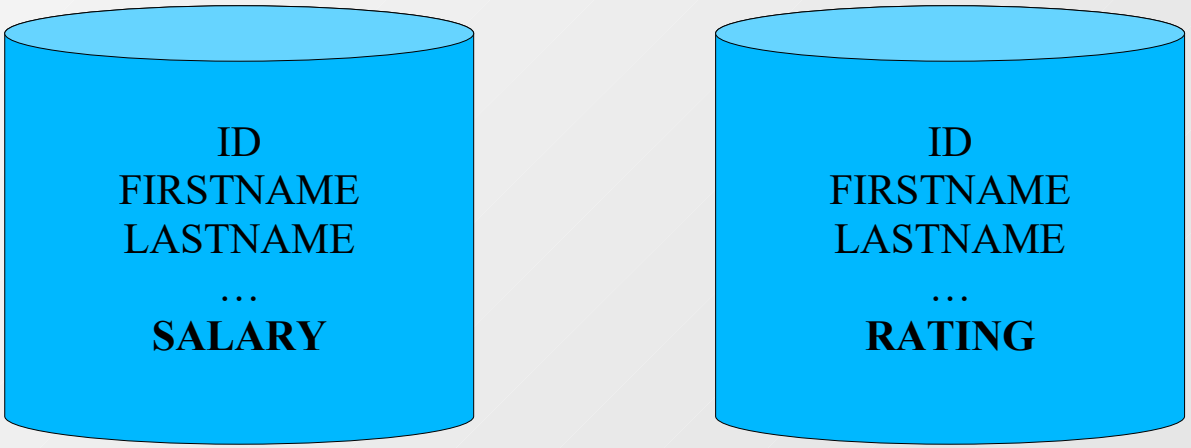
```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
public abstract class Person implements Serializable {
    …
}
```



Der Diskriminator-Wert kann mittels Annoation @DiskriminatorValue konfiguriert werden (ansonsten gleich Klassenname)!

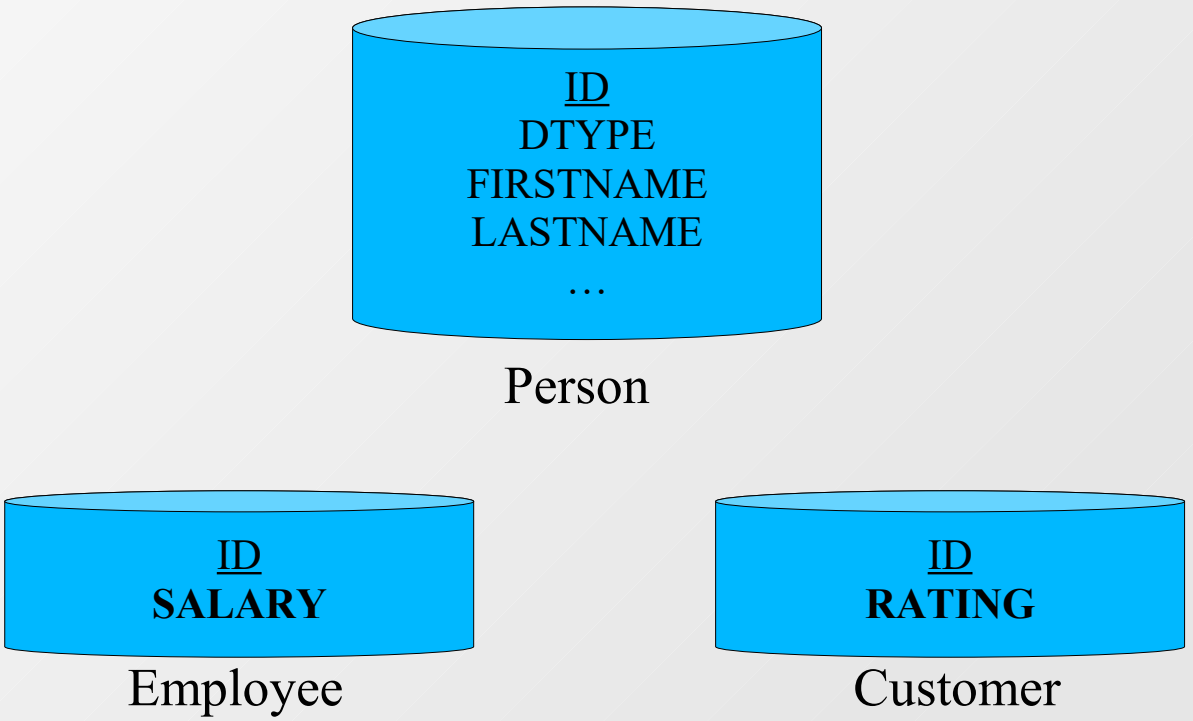
Vererbung – TABLE_PER_CLASS

Es wird eine **Tabelle je konkreter Klasse** angelegt, die alle Attribute der jeweiligen Klasse enthält.



Vererbung – JOINED

Die Attribute der Basisklasse werden in einer Tabelle gespeichert und die Attribute der abgeleiteten Klasse in je einer weiteren Tabelle mit gleichem Key.
Der Diskriminator (DTYPE) in der Tabelle der Basisklasse hilft bei der Unterscheidung.



Beziehungen

Die grundlegenden Beziehungsarten (1:1, 1:n, n:m) können über folgende Annotationen gesteuert werden. Details dazu in den Übungen!

- @OneToMany(cascade=CascadeType.ALL)
  @JoinColumn(name="xyz")
  [Collection Attribut]

- @ManyToOne
  [Attribut]

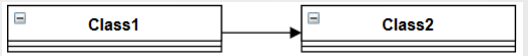
- mappedBy-Attribut: Gibt an, dass die Beziehung bereits auf der Gegenseite definiert wurde.

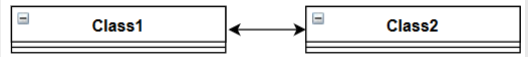
Beziehungen

Es existieren 3 grundlegende ***Arten von Beziehungen***:

- 1:1
- 1:n
- n:m (Assoziativtabelle notwendig)

Möglichkeiten zur ***Navigation im Klassenmodell***:

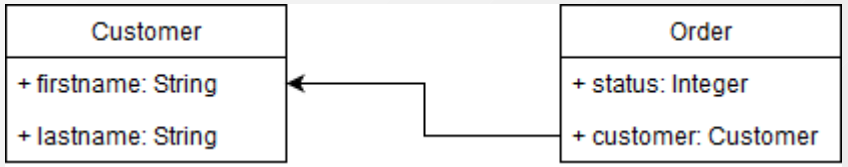
- unidirektional 

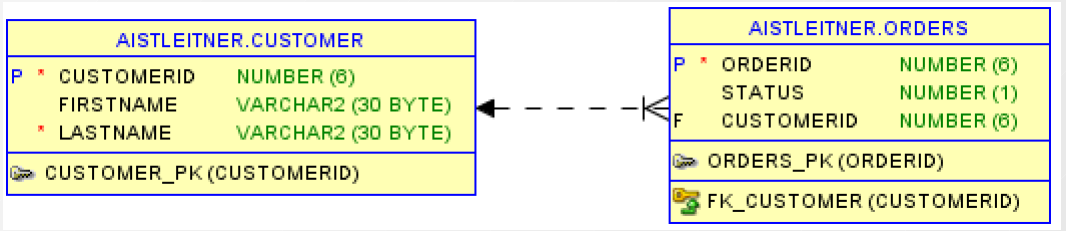
- bidirektional 

Realisierung im ***Datenmodell***:

- Mit Assoziativtabelle (bei 1:n, n:m)
- Ohne Assoziativtabelle (nur bei 1:1 und 1:n)

Beziehungen – 1:n ohne assoz. Tabelle

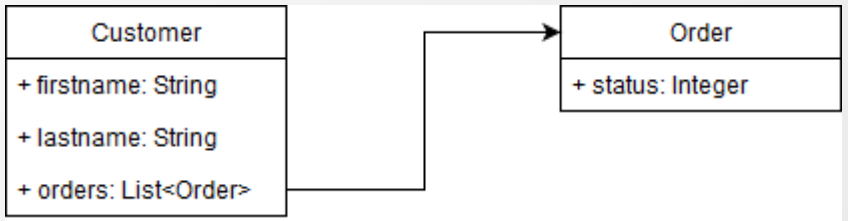


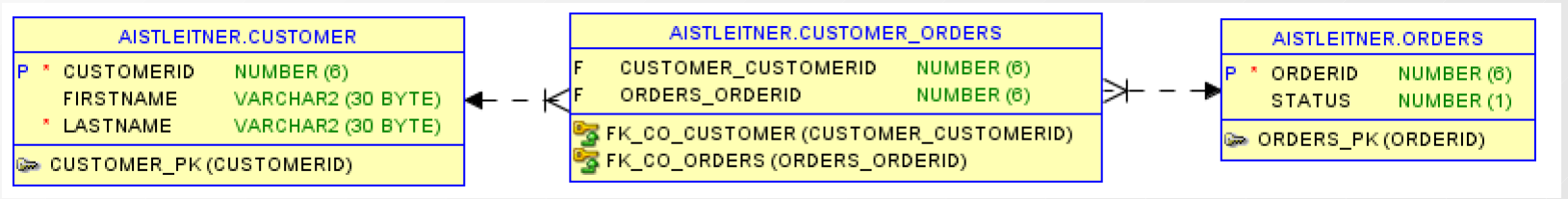



```java
@Entity
@SequenceGenerator(name="custid_seq",
public class Customer {

    @Id
    @GeneratedValue(generator = "cust:
    int customerId;

    String firstname;
    String lastname;
```

```java
@Entity
@SequenceGenerator(name="order_seq", initialV
@Table(name = "ORDERS")
public class Order {

    @Id
    @GeneratedValue(generator = "order_seq",
    int orderId;

    int status;

    @ManyToOne
    @JoinColumn(name = "customerId")
    Customer customer;
```

Beziehungen – 1:n mit assoz. Tabelle

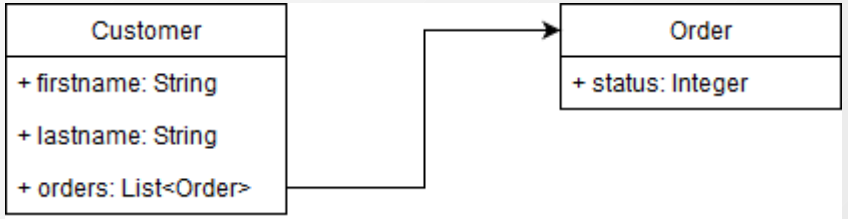


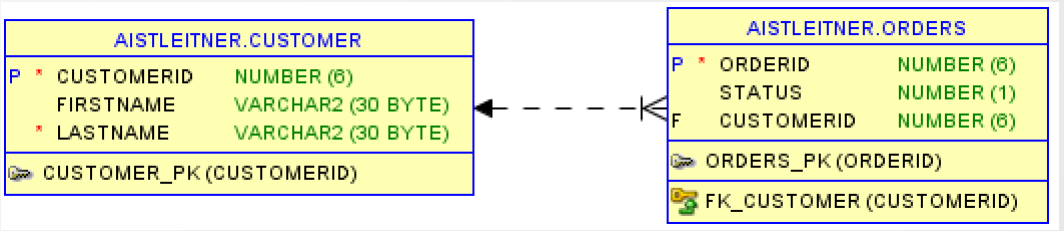



```
@Entity
@SequenceGenerator(name="custid_seq", initial
public class Customer {

    @Id
    @GeneratedValue(generator = "custid_seq",
    int customerId;

    String firstname;
    String lastname;

    @OneToMany(cascade = CascadeType.ALL)
    List<Order> orders;
```

```
@Entity
@SequenceGenerator(name="order_
@Table(name = "ORDERS")
public class Order {

    @Id
    @GeneratedValue(generator =
    int orderId;

    int status;

    public int getOrderId() {
        return orderId;
    }
```

Beziehungen – 1:n ohne assoz. Tabelle (Möglichkeit 2)

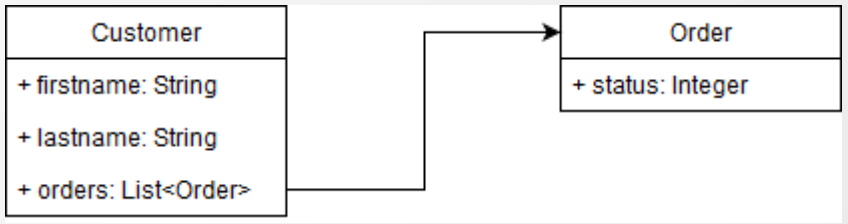


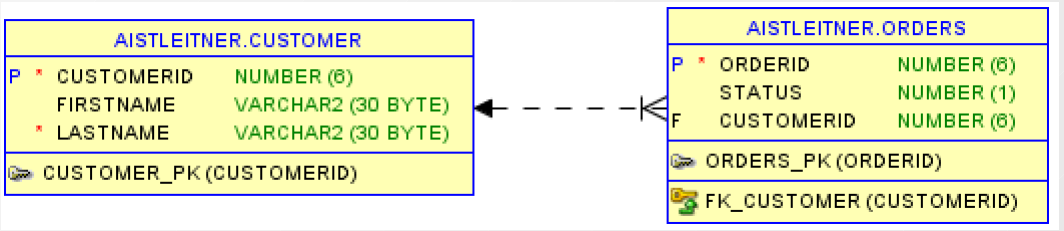



```java
@Entity
@SequenceGenerator(name="custid_seq", ini
public class Customer {

    @Id
    @GeneratedValue(generator = "custid_s
    int customerId;

    String firstname;
    String lastname;

    @OneToMany(cascade = CascadeType.ALL)
    @JoinColumn(name="customerId")
    List<Order> orders;
```

```java
@Entity
@SequenceGenerator(name="order_
@Table(name = "ORDERS")
public class Order {

    @Id
    @GeneratedValue(generator =
    int orderId;

    int status;

    public int getOrderId() {
        return orderId;
    }
```

Beziehungen – 1:n bidirektional ohne assoz. Tabelle






```java
@Entity
@SequenceGenerator(name="custid_seq", ini
public class Customer {

    @Id
    @GeneratedValue(generator = "custid_s
    int customerId;

    String firstname;
    String lastname;

    @OneToMany(cascade = CascadeType.ALL,
               mappedBy="customer")
    List<Order> orders;
```

```java
@Entity
@SequenceGenerator(name="order_seq",
@Table(name = "ORDERS")
public class Order {

    @Id
    @GeneratedValue(generator = "orde
    int orderId;

    int status;

    @ManyToOne
    @JoinColumn(name="customerId")
    Customer customer;
```

Beziehungen – Eager und Lazy Loading

JPA bietet die Möglichkeit, im Mapping anzugeben, ob eine Beziehung gleich mitgeladen werden soll, wenn die eigentliche Entität geladen wird.
Dies ist möglich, indem man im Mapping den ***`FetchType.EAGER`*** angibt.

Dieser ist bei To-one-Beziehungen default.
Bei To-many-Beziehungen dagegen muss er explizit angegeben werden.

Die Alternative ist ***`FetchType.LAZY`***.
Die Daten werden erst beim Zugriff auf die Felder nachgeladen.

Embeddable

Arbeiten mehrere Entity-Klassen mit den gleichen Attributen, können diese Attribute in eine mit ***@Embeddable*** annotierte Klasse ausgelagert werden.

Diese wird dann mittels ***@Embedded*** eingebunden.

Dies ist auch für Composite-Keys möglich, die dann mit ***@EmbeddedId*** annotiert werden.