



Datenbanken 2

– Kapitel 2: Java Persistence API –



Java Persistence API – JPA

Inhalte des Kapitels

- Überblick über die Entwicklung der EJB- und JPA-Spezifikationen
- JPA – Persistenz Teil 1
- JPA – Entity-Klassen (Entities)
- JPA – Beziehungen
- JPA – Persistenz Teil 2

Lernziele

- Entwicklung der EJB- und JPA-Spezifikationen zeitlich einordnen können und die Referenz-Implementierungen kennen
- Grundlegende Konzepte des ORM bzgl. der Annotationen von
 - Entity-Klassen,
 - unterschiedlichsten Beziehungen und der (Collection-)Datentypen, insbesondere von mehrwertigen Referenzattributen, sowie
 - Persistierungkennen und anwenden können.



EJB – Enterprise Java Beans

- **EJB 1.0 – Final Specification 1997**
 - serverseitiges Komponentenmodell, mit dem (verteilte) Geschäftsobjekte entwickelt und dann durch **EJB-Container** verwaltet werden.
 - **Komponenten = Enterprise Beans:** Session Beans, Entity Beans, Messagedriven Beans
- **EJB 1.1 – erste Verbesserung 1999, zusammen mit J2EE**
 - Container- und BeanManaged Persistence (CMP, BMP)
 - noch keine Unterstützung von Beziehungsfeldern
- **EJB 2.0 – 2001**
 - Neuerungen in der Container Managed Persistence (CMP), die nicht abwärtskompatibel sind
- **EJB 2.1 – 2002**
 - zustandslose Session-Beans (Web Services)
 - Erweiterungen JPQL (Aggregatfunktionen, ...)



EJB 3.0 und die Java Persistence API

- **EJB 3.0 – JSR 220 – Final Release 2006**
 - Ziel: Vereinfachung von Java EE
 - ist Bestandteil der **Java Enterprise Edition 5 Plattform (JEE)**
 - **EJB 3.1** 2008 ab **Java EE 6**, **EJB 3.2** – JSR 345 2013 ab **Java EE 8**
-

⇒ **Java Persistence API – JPA 1.0**

- eigenständiger Bestandteil der EJB-3.0-Spezifikation
 - löst die Entity Beans ab
 - **Java Persistence Query Language – JPQL**
 - wurde im Vergleich zu EJB 2.0 erweitert
- **JPA 2.0 – JSR 317 – Final Release 2009**
 - ist Bestandteil der Java Enterprise Editionen (JEE) ab Version 6 • aktuell: Java EE-Spezifikation Version 7, Mai 2013
 - Ergänzung verschiedener Feature sowohl im Bereich der Funktionalität (u.a. ElementCollection) als auch im Bereich Performance (u.a. Locking Strategien)
 - Integration diverser proprietärer Erweiterungen in den Standard
 - **JPA 2.1** – JSR 338 seit April 2013



Java Persistence API – JPA

- Darstellung der Entitäten erfolgt durch **POJO – Plain old Java Objects**.
 - Zur **Definition der Metadaten** werden Java **Annotations** (seit Java 5.0) verwendet. Zuvor verwendete XML Deskriptor-Dateien können nach wie vor alternativ verwendet werden.
- Die JPA ist nicht auf den Einsatz unter Java EE begrenzt, sondern kann auch unter Java SE, also außerhalb eines Java EE Containers, eingesetzt werden.
- Es handelt sich bei der JPA um eine Spezifikation und NICHT um ein fertiges Framework!
- Implementierungen u.a. Hibernate, Oracle TopLinkEssentials¹, EclipseLink², Apache OpenJPA, Bea Kodo, ...
- 1) Referenzimplementierung für JPA 1.0
2) Referenzimplementierung ab JPA 2.0



Datenbanken 2

✓ Einführung

2. Java Persistence API

→ Persistenz Teil 1

- Entity-Klassen
- Beziehungen
- Persistenz Teil 2

3. Datenbankabfragen

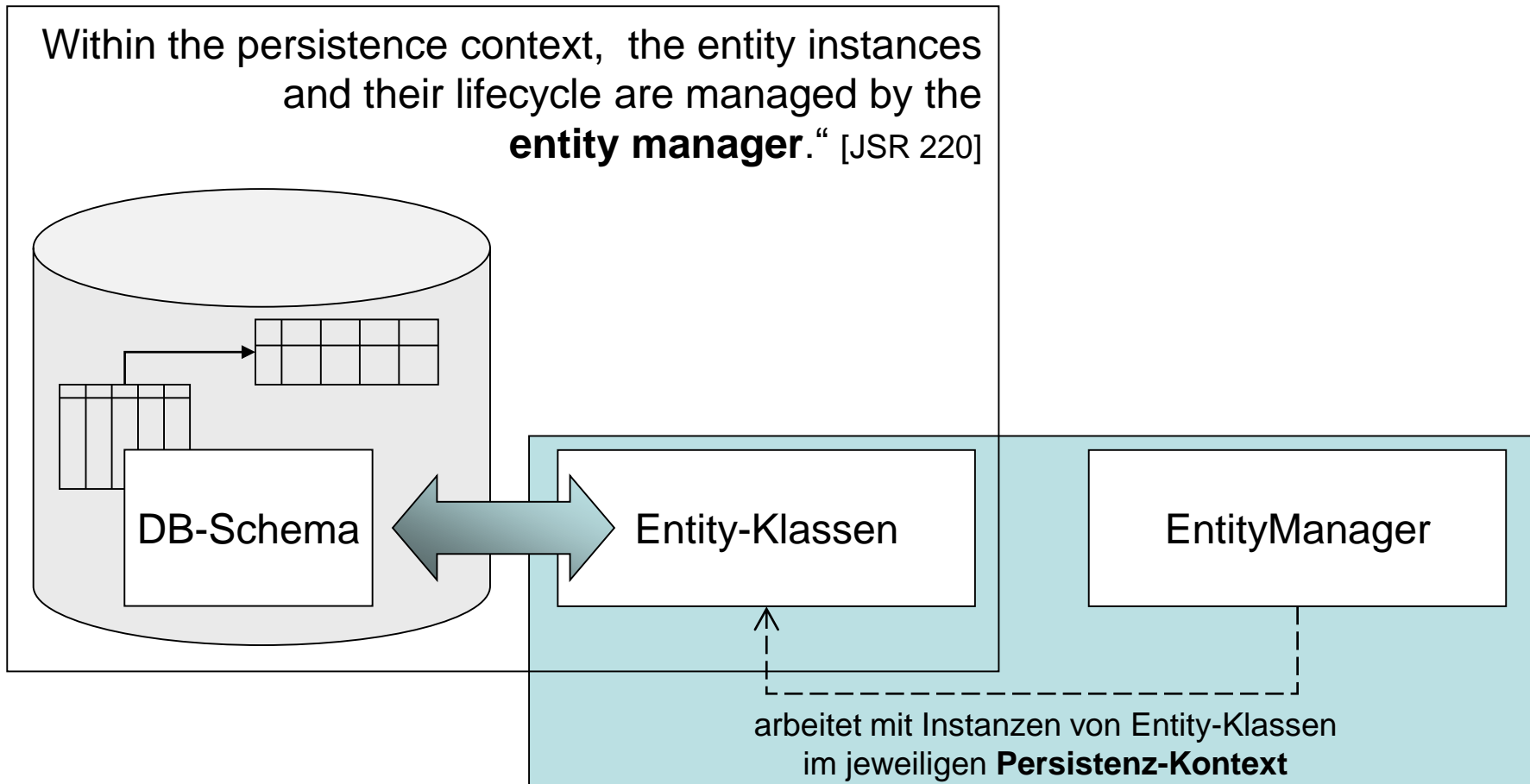
4. Transaktionsmanagement, Caching und Ladestrategien

5. Pufferverwaltung und Optimierung von Zugriffspfaden



JPA – Persistenzkontext

„A **persistence context** is a set of managed entity instances in which for any persistent entity identity there is a unique entity instance.“





JPA – Entity Manager

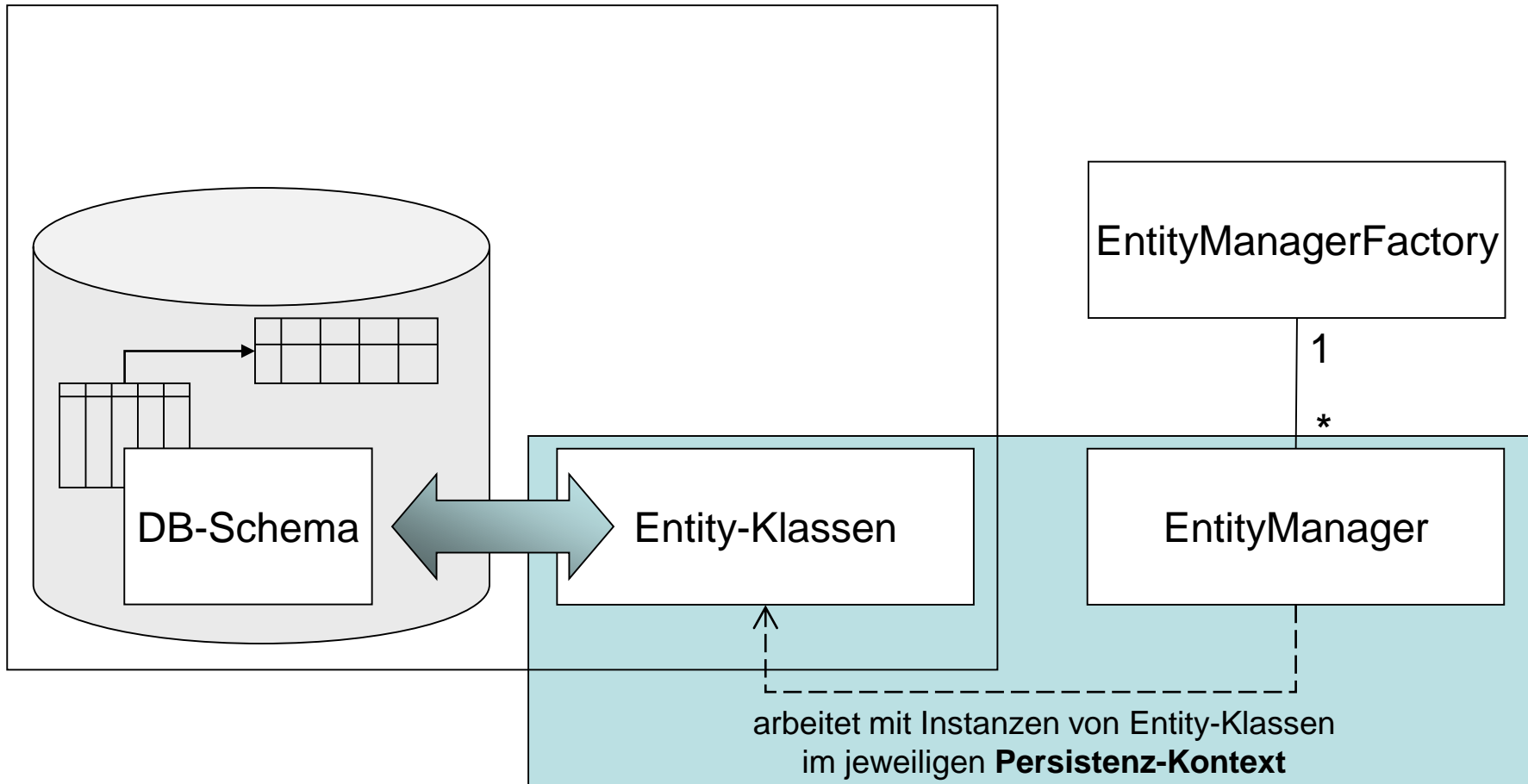
- Die Verwaltung der Entities erfolgt durch einen **Entity Manager**: er erzeugt, ändert, löscht und sucht Entity-Instanzen.
 - **application managed** Entity Manager – von der Anwendung selbst verwaltet → Anwendung im Praktikum!
 - **container managed** Entity Manager – nur für Java EE Container
- Der **Persistenzkontext** umfasst eine Menge von Entity-Instanzen, die zu jedem Datensatz einer Tabelle der Datenbank jeweils höchstens ein Java-Objekt enthält.
- Gültigkeitsdauer eines Persistenzkontext:
 - **RESOURCE_LOCAL**: *application-managed (default)* bei lokalen Ressourcen
 - **TRANSACTION**: Gültigkeitsdauer = Dauer einer Transaktion, *container-managed (default)*. Persistenzkontext wird nach Ablauf einer Transaktion explizit über den Entity Manager geschlossen
 - **EXTENDED**: Gültigkeitsdauer kann mehrere Transaktionen umfassen – *container-managed*. Persistenzkontext wird über mehrere Transaktionen offen gehalten.

Ein Entity Manager ist immer genau einem Persistenzkontext zugeordnet.



JPA – Entity Manager Factory

- Entity Manager-Instanzen werden von der Instanz einer **Entity Manager Factory** zur Verfügung gestellt.





JPA – Entity Manager Factory

Eine **EntityManagerFactory**

- hält in ihrem Cache Mappingdaten und die generierten SQL-Anweisungen und
- kann für die Anwendung EntityManager-Instanzen erzeugen ...

```
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

...
private static EntityManagerFactory emf;
private static EntityManager em;

...
// Die EntityManagerFactory erstellen
emf = Persistence.createEntityManagerFactory("MyProjectPU");

// Neuen EntityManager erstellen
em = emf.createEntityManager();

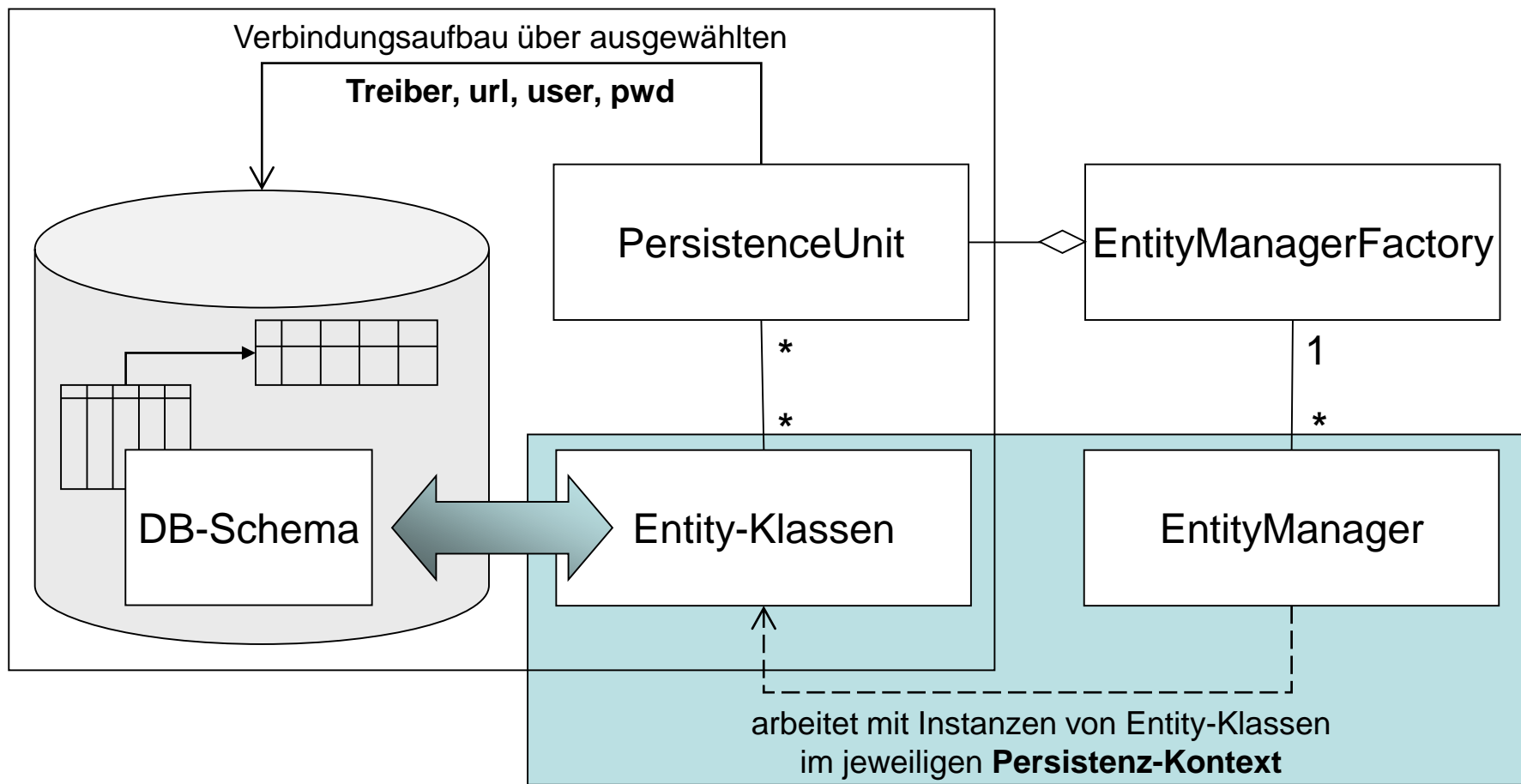
...
```

... wobei der Name einer zugehörigen **PersistenceUnit** übergeben werden muss (im Beispiel "MyProjectPU"):



JPA – Persistence Unit

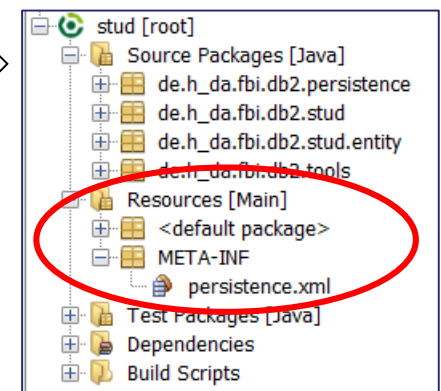
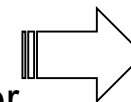
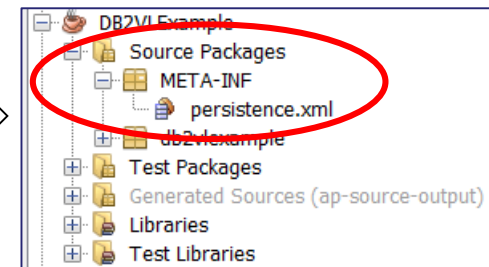
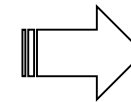
„A **persistence unit** defines the set of all classes that are related or grouped by the application and which must be colocated in their mapping **to a single database**.“ [JSR 220]





JPA – Persistence Unit

- Eine **Persistence Unit** ist eine logische Einheit, der
 - eine **EntityManagerFactory** mit deren
 - zugehörigen **EntityManagern** zugeordnet wird, ebenso alle
 - mit **@Entity** versehenen Klassen und
 - deren **Mappingkonfigurationen**.
- Eine Persistence Unit wird über die xml-Datei **persistence.xml** beschrieben, die üblicherweise im Package META-INF als Unterpackage von **Source Packages** liegt (so in unseren Beispiel-Projekten).
- Durch die Nutzung von Gradle zur Unterstützung des CI-Prozesses auf unserem git-Server liegt sie für unser Praktikum im Package META-INF als Unterpackage von **Resources [Main]**.
- In der Konfigurationsdatei persistence.xml können beliebig viele Persistence Units definiert werden.





JPA – Persistence Unit

Zahlreiche Eigenschaften werden in der `persistence.xml` gesetzt, insbesondere die **Verbindungseigenschaften** :

- `javax.persistence.jdbc.user` → Username Datenbank(schema)
- `javax.persistence.jdbc.passwort` → Passwort Datenbank(schema)
- `javax.persistence.schema-generation.database.action` → Art der Schema-Erzeugung

Beispiel für eine `persistence.xml`:

```
<persistence-unit name="MyProjectPU" transaction-type="RESOURCE_LOCAL">
  <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
  <properties>
    <property name="javax.persistence.jdbc.driver" value="oracle.jdbc.OracleDriver"/>
    <property name="javax.persistence.jdbc.url"
      value="jdbc:oracle:thin:@localhost:1521:xe"/>
    <property name="javax.persistence.jdbc.user" value="praktikum"/>
    <property name="javax.persistence.jdbc.password" value=" " />
    <property name="javax.persistence.schema-generation.database.action"
      value="drop-and-create"/>
  </properties>
</persistence-unit>
```



Table Generation Strategy

- Als weiteres wichtiges Property wird in der persistence.xml die `schema-generation.database.action` festgelegt, in der die Table Generation Strategy beschrieben wird :

Table Generation Strategy: ☐ Create ☒ Drop and Create ☐ None

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

→ Die drei möglichen Attributwerte haben die folgende Bedeutung:

- Drop and Create:** Alle zu generierenden Tabellen werden vor dem Neuanlegen mit Drop aus dem Schema gelöscht, danach entsprechend der Informationen in den Entity-Klassen neu angelegt.
 - *Achtung: Tabellen gelöschter Entity-Klassen müssen manuell aus dem Datenbankschema gelöscht werden!*
- Create:** Nur neu hinzukommende Tabellen werden neu angelegt, die vorhandenen Tabellen bleiben unverändert.
- None:** Das Datenbankschema wird nicht verändert.



Logging von eclipseLink sql-Anweisungen

- Zum Logging der sql-Anweisungen, die von eclipseLink zum Datebank-Server geschickt werden, können in der **persistence.xml** die folgenden provider-spezifischen Eigenschaften gesetzt werden:

```
- <properties>
  ...
  <property name="eclipselink.logging.level.sql" value="FINE" />
  ...
</properties>
```

Output - MyProject (run) persistence.xml

```
[EL Fine]: Connection(823458691)--INSERT INTO borrower (borrower_id, borrower_name) VALUES (?, ?)
bind => [152, Schestag]
nach dem Hinzufügen zum Persistenzkontext
[EL Fine]: Connection(823458691)--INSERT INTO AUDIOBOOK (book_id, SPEAKER, title, borrower_id) VALUES (?, ?, ?, ?)
bind => [153, Otto, Titel 2 - ohne Entleiher, null]
[EL Fine]: Connection(823458691)--INSERT INTO book (book_id, title, borrower_id) VALUES (?, ?, ?)
bind => [151, Titel 1 - mit Entleiher, 152]
[EL Fine]: Connection(823458691)--INSERT INTO PAPERBACK (book_id, NUMBEROFPAGES, title, borrower_id) VALUES (?, ?, ?, ?)
bind => [154, 345, My Paperback, 152]
[EL Fine]: Connection(823458691)--SELECT t0.borrower_id, t0.borrower_name, t1.book_id, t1.title, t1.borrower_id FROM book t1 LEFT OUTER JOIN borrower t0 ON t1.borrower_id = t0.borrower_id
```



JPA – Persistenz: Beispiel (Code)

- ... für die Verwendung eines Entity Managers:

```
// Neuen EntityManager erstellen
EntityManager em = emf.createEntityManager();
// Transaktion starten
em.getTransaction().begin();
Kunde kunde1 = em.find(Kunde.class, new Long(123));
kunde1.setName("Schmidt"); // verwaltete Entity
Kunde kunde2 = new Kunde();
kunde2.setName("Meyer"); // neue Entity
em.persist(kunde2); // verwaltete Entity
em.getTransaction().commit();
em.close();
```

emf = Instanz der
EntityManagerFactory

em.find(...) lädt eine
Kundeninstanz aus der DB
in den Persistenzkontext *)

kunde1.setName(...) weist
der Kundeninstanz im Per-
sistenzkontext einen neuen
Namen zu.

kunde2 wird instanziiert
und Werte werden zuge-
wiesen.

persist(kunde2) über-
nimmt kunde2 in den Per-
sistenzkontext.

commit() macht alle Ände-
rungen persistent.

*) wenn die Instanz im Persistenzkontext nicht schon vorhanden ist.



JPA – Persistenz: Beispiel (Datensicht)

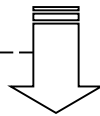
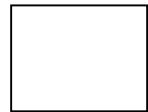
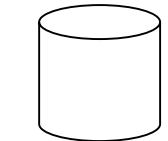
- ... für die Verwendung eines Entity Managers:

Kunde

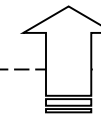
Id	Name	
123	Schmitt	

Kunde

Id	Name	...
123	Schmidt	...
124	Meyer	...



em.find(...);



em.commit();

Persistenzkontext von em

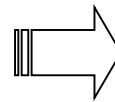
123 | ~~Schmitt~~ Schmidt

kunde1.setName(...);

NULL | Meyer

Kunde kunde2 = new Kunde();

...



em.persist(kunde2);

Persistenzkontext von em

123 | Schmidt

124 | Meyer



Datenbanken 2

✓ Einführung

2. Java Persistence API

✓ Persistenz Teil 1

→ Entity-Klassen

- Beziehungen
- Persistenz Teil 2

3. Datenbankabfragen

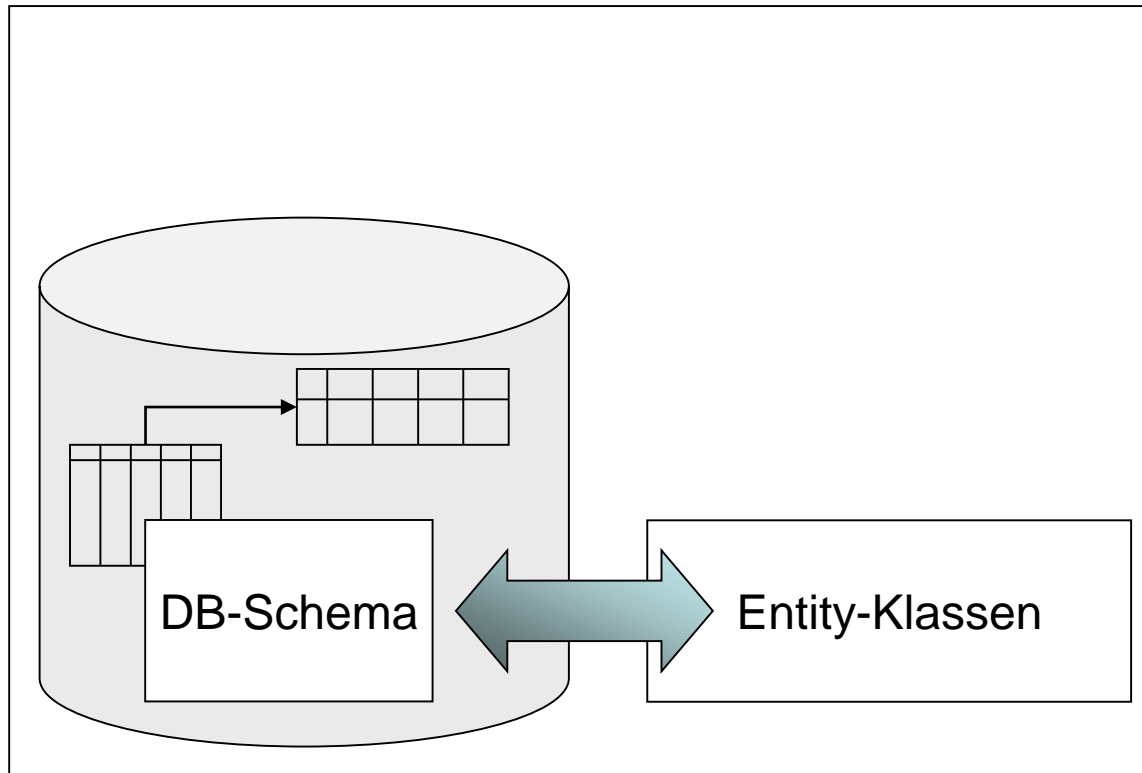
4. Transaktionsmanagement, Caching und Ladestrategien

5. Pufferverwaltung und Optimierung von Zugriffspfaden



JPA – Entity-Klassen (Entities)

„An entity is a lightweight persistent domain object.“ [JSR 220]



→ Entity-Klassen entsprechen den Objekten, die durch das DB-Schema beschrieben werden – alle zum Mapping notwendigen Metainformationen des DB-Schemas können über Annotationen beschrieben werden:



JPA – Entity-Klassen (Entities)

- Entity-Klassen können abstrakte oder konkrete Java-Klassen sein, sie müssen nicht abgeleitet sein. Entities und andere Java-Klassen können innerhalb einer Vererbungshierarchie beliebig kombiniert werden.
- Vererbung, polymorphe Abfragen und polymorphe Assoziationen werden von Entities unterstützt.

Bedingungen an Entity-Klassen

- Markierung mit **Annotation @Entity** ist erforderlich.
- Ein **parameterloser Konstruktor** (public oder protected) muss enthalten sein (implizit oder explizit).
- Sie dürfen **nicht als final deklariert** sein, ebenso nicht ihre Methoden und die persistenten Attribute.
- Sie müssen einen **Primärschlüssel** enthalten (**Annotation @Id**)
 - Primärschlüssel: einfaches Attribut oder zusammengesetzter Schlüssel (etwas komplizierter in der Realisierung)
- Persistente Attribute können durch **Annotationen** (direkt oder bei ihren Getter-Methoden) Mapping- und andere Meta-Informationen erhalten.



JPA – Entity-Klassen (Entities)

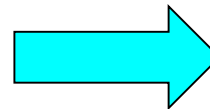
- Einfache Entity-Klasse (noch ohne Beziehungen)

@Entity

```
public class Kunde {  
    @Id  
    private int id;  
    private String name;  
    public Kunde() {} // Default Konstruktor  
    // evtl. weitere Konstruktoren  
    ...  
    //Getter- und Setter-Methoden  
    ...  
}
```

- Wenn die Persistenz-Umgebung eingerichtet ist (s. Abschnitte **Persistenz 1 und 2** in diesem Kapitel), kann man ein Objekt der Entity-Klasse Kunde (innerhalb einer Transaktion!) in die Datenbank schreiben.

```
...  
Kunde kunde = new Kunde();  
Kunde.setId(123);  
kunde.setName("Testkunde");  
em.persist(kunde);  
...
```



	ID	NAME
1	123	Testkunde



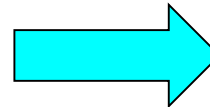
JPA – Primärschlüssel (1)



- Häufig verwendet man bei der Verwendung von Objekt-relationalen Mapping-Frameworks synthetische Primärschlüssel.

```
@Entity
public class Kunde {
    @Id
    @GeneratedValue
    private int id;
    private String name;
    public Kunde() {} // Default Konstruktor
    ...
}
```

- Der Primärschlüssel darf dann nicht(!) im Programm gesetzt werden.

```
...
Kunde kunde = new Kunde();
Kunde.setId(123);
kunde.setName("Testkunde");
em.persist(kunde);
...
```



	 ID	 NAME
1	1	Testkunde



JPA – Primärschlüssel (2)

- Die Strategie, wie der synthetische Primärschlüssel generiert werden soll, kann bei Bedarf noch genauer spezifiziert werden:
- **Generatorstrategien der JPA**
 - AUTO: entsprechend der darunterliegenden DBMS-Strategie
 - TABLE: IDs in eigener Table
 - IDENTITY: unterstützt Identity Columns, z.B. *autoincrement* in MySQL
 - SEQUENCE: unterstützt Sequences, z.B. in Oracle, PostgreSQL

@Entity

```
public class Kunde {
```

@Id

```
    @GeneratedValue(strategy = GenerationType.SEQUENCE)
```

```
    private int id;
```

```
    ...
```

```
}
```

- Weitere proprietäre Generatorstrategien finden sich bei den unterschiedlichen OR-Mappern.



hashCode() und equals()

- Java-Klassen besitzen zu Vergleichszwecken zwei Methoden der Rootklasse Object aus der Java Klassenhierarchie: equals() und hashCode().
 - equals(Object obj) hat den Returntyp boolean, wobei true bedeutet, dass die Objekte gleich, false, dass sie nicht gleich sind.
 - Gleichheit kann beliebig definiert werden, aber es muss immer gelten: $a.equals(b) == b.equals(a)$
 - Bei Entity-Instanzen gilt: Objekte sind gleich, wenn der Wert des Id-Attributes (Primärschlüssel im Datenbankschema) gleich ist.
 - hashCode() liefert einen hash-Wert als int zurück, welcher zum Test verwendet wird, ob Objekte gleich sein „könnten“.
 - Es gilt $a.equals(b) \Rightarrow a.hashCode() == b.hashCode()$
 - Es gilt jedoch nicht(!) notwendig $a.hashCode() == b.hashCode() \Rightarrow a.equals(b)$
 - Der hashCode wird bei Entities aus dem Primärschlüsselwert gebildet.
- NetBeans ist in der Lage, die Methoden equals() und hashCode() automatisch für Entity-Klassen zu generieren → vgl. Praktikum.



Temporal-Annotationen

- Eigenschaften, die Datumswerte repräsentieren, müssen in JPA durch die Annotation `@Temporal` gekennzeichnet werden.
- Java kannte bis vor Kurzem nur den Datentyp `Date` (= Datum und Uhrzeit), während SQL die drei Datentypen `date`, `time`, `timestamp` kennt.
- Beispiel für einen Timestamp:

```
@Entity
public class Kunde {
    ...
    private Date lastOrderDate;
    @Temporal(strategy = GenerationType.DATE)
    Public Date getLastOrderDate() {
        return lastOrderDate;
    }
    ...
}
```

- Mit Java 8 wurde die Java-API um diverse Datentypen für Datum und Uhrzeit ergänzt. Diese sind jedoch nicht überall in JPA ohne Erweiterungen unterstützt → Details hierzu s. Literatur bzw. Praktikum.



Datenbanken 2

✓ Einführung

2. Java Persistence API

✓ Persistenz Teil 1

✓ Entity-Klassen

→ Beziehungen: Assoziationen, Komposition/Aggregation, Vererbung

- Persistenz Teil 2

3. Datenbankabfragen

4. Transaktionsmanagement, Caching und Ladestrategien

5. Pufferverwaltung und Optimierung von Zugriffspfaden



JPA – Assoziationen

- Die JPA unterstützt die folgenden Beziehungstypen mit ihren Annotationen:
 - 1:1 mit der Annotation @OneToOne,
 - 1:n mit der Annotation @OneToMany,
 - n:1 mit der Annotation @ManyToOne,
 - n:m mit der Annotation @ManyToMany.

Relationenmodell

- In der Regel sind nur unidirektionale Beziehungen möglich – außer im Fall einer 1:1-Beziehung.

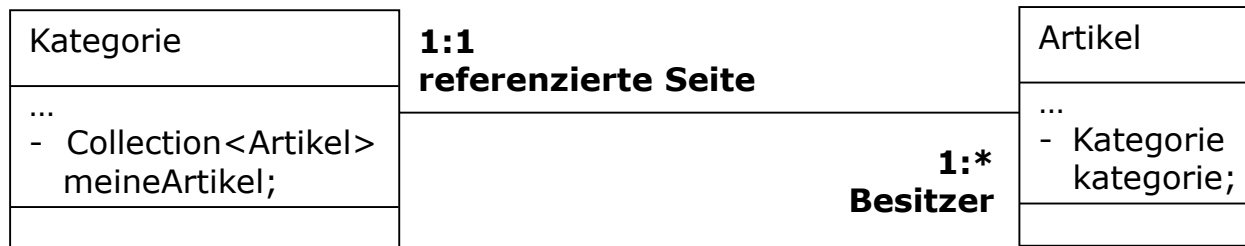
JPA

- Es werden uni- und bidirektionalen Beziehungen unterstützt.
- Nachteile unidirektionaler Beziehungen im Anwendungsprogramm?
- Bei bidirektionalen Beziehungen gibt es genau einen „**Besitzer**“ und genau eine „**referenzierte Seite**“:



JPA – Bidirektionale 1:n-Beziehungen (1)

- Diejenige Entity, die die **referenzierte Seite** repräsentiert, muss auf ihren „Besitzer“ durch die Angabe des Feldes **mappedBy** der entsprechenden Relationship-Annotation verweisen.
- Die „**n**“-**Seite** einer bidirektionalen Beziehung muss die „**Besitzer**“-**Seite** sein (der „Besitzer“ des Fremdschlüssels in der Datenbank!).



```
@Entity
public class Kategorie ... { ...
    private Collection<Artikel> meineArtikel =
        new HashSet();
    @OneToMany(mappedBy = "kategorie")
    public Collection<Artikel> getMeineArtikel() {
        return meineArtikel;
    }
    ...
}
```

Wert des mappedBy-Attributes = Name
des Referenzattributes in der Besitzer-Klasse.

```
@Entity
public class Artikel ... {
    ...
    private Kategorie kategorie;
    @ManyToOne
    public Kategorie getKategorie() {
        return kategorie;
    }
    ...
}
```



JPA – Bidirektionale 1:n-Beziehungen (2)

Auswirkung der Kennzeichnung durch das mappedBy-Attribut:

- Wird mit neuen (oder bestehenden) Instanzen eine neue Beziehung implementiert, so muss dies jeweils für beide Referenzattribute erfolgen:



```
Kategorie neueKategorie = new Kategorie();
```

```
// Diese Zuordnung alleine würde auf der DB NICHT das  
// Setzen des FK-Wertes in der Artikeltabelle bewirken:
```

```
neueKategorie.getMeineArtikel().add(neuerArtikel) ;
```

```
Artikel neuerArtikel = new Artikel();
```

```
// Nur diese Zuordnung bewirkt eine  
// Synchronisation mit der Datenbank:  
neuerArtikel.setKategorie(neueKategorie);
```

- Das mappedBy-Attribut wird auch als „inverses“ Attribut bezeichnet.
Diese Seite spielt keine Rolle bei der Synchronisation mit der DB!



JPA – Spezifikation und „Wirklichkeit“ (1)

... am Beispiel einer **unidirektionalen 1:n-Beziehung**, implementiert in Richtung **@OneToMany**:

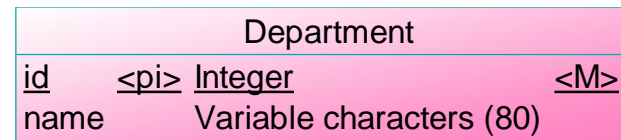
- Wer ist Owner, wer ist „referenzierte Seite“ dieser Implementierung?

@Entity

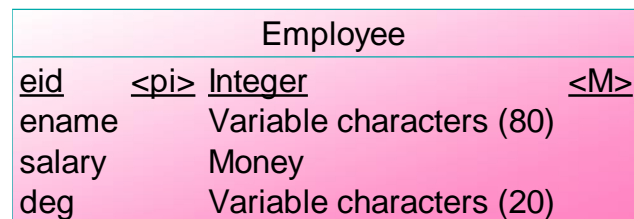
```
public class Department {  
    @Id  
    ...  
    private int id;  
    private String name;  
    @OneToMany  
    private List <Employee> employeelist;  
}
```

@Entity

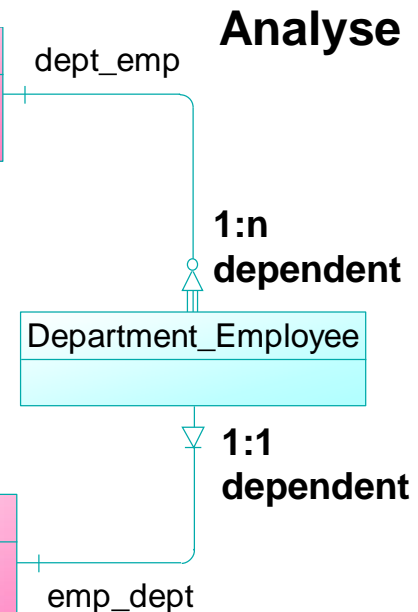
```
public class Employee{  
    @Id  
    ...  
    private int id;  
    private String name;  
    private double salary;  
    private String deg;  
}
```



Owner



*„eigentlich“ referenzierte Seite,
aber keine mengenwertige Referenz
von Department auf Employee möglich!*





JPA – Spezifikation und „Wirklichkeit“ (2)

→ Welche zusätzliche Spalten-Eigenschaft muss in der Tabelle Department_Employee deklariert werden um zu garantieren, dass jeder Employee nur **genau einem** Department zugeordnet wird?

Design

DEPARTMENT		
<u>ID</u>	INTEGER	<pk>
NAME	VARCHAR2(80)	

DEPARTMENT_EMPLOYEE			
<u>DEPARTMENT_ID</u>	INTEGER	<pk,fk1>	
<u>EMPLOYEE_EID</u>	INTEGER	<pk,fk2>	

→ "... *There is a unique key constraint on the foreign key that refers to table EMPLOYEE.*"

[JSR338: JPA 2.1: 2.10.5.1 Unidirectional OneToMany Relationships]

EMPLOYEE		
<u>EID</u>	INTEGER	<pk>
ENAME	VARCHAR2(80)	
SALARY	NUMBER(8,2)	
DEG	VARCHAR2(20)	

ID = DEPARTMENT_ID

EID = EMPLOYEE_EID



JPA – Spezifikation und „Wirklichkeit“ (3)

- Die Implementierung gemäß Spezifikation wird in EclipseLink nicht vollständig umgesetzt:
Es fehlt der Uniqueness-Constraint auf der *fk2*-Spalte!
 - De Facto bedeutet dies, dass diese Implementierung einer m:n-Implementierung entspricht und einer Employee-Instanz beliebig viele Department-Instanzen zugeordnet werden können – ein Fehler!
- Auch bei korrekter Implementierung gemäß Spezifikation ist die Zwischentabelle im Schema aus Performancegründen ungünstig – mehr Joins als „eigentlich“ notwendig!
- Es empfiehlt sich zur Vermeidung dieses Effektes **in jedem Fall eine Implementierung der Beziehung in Richtung @ManyToOne**, sodass diese Seite Owner der Beziehung ist, auch wenn die Implementierung der Richtung @OneToMany zur Unterstützung der applikatorischen Prozesse letztlich ausreichend wäre:



JPA – Spezifikation und „Wirklichkeit“ (4)

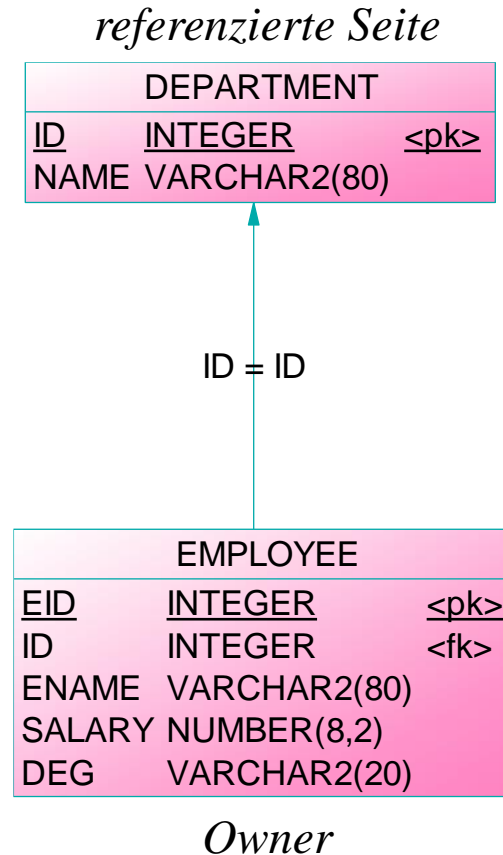
Effiziente Implementierung der 1:n-Beziehung, bidirektional:

@Entity

```
public class Department {  
    @Id  
    ...  
    private int id;  
    private String name;  
    @OneToMany (mappedBy = "dept")  
    private List <Employee> employeeelist;
```

@Entity

```
public class Employee{  
    @Id  
    ...  
    private int id;  
    private String name;  
    private double salary;  
    private String deg;  
    @ManyToOne  
    private Department dept;
```





JPA – Anmerkungen zur Annotation (1)

- Die Relationship kann annotiert werden
 - am **Referenzattribut** (= Standard beim NetBeans Re-Engineering):

@OneToMany (mappedBy = "kunde")	Annotation des Referenzattributes
private Collection<Bestellung> bestellungen = new HashSet();	

- an der **get-Methode des Referenzattributes**:

private Collection<Bestellung> bestellungen = new HashSet();	
@OneToMany (mappedBy = "kunde")	
public Collection<Bestellung> getBestellungen() {	
return bestellungen;	
}	
public void setBestellungen (Collection<Bestellung> b) {	
this.bestellungen = b ;	
}	
	Annotation der get-Methode für das Referenzattribut

- Im zweiten Fall **muss** die set-Methode für das Referenzattribut implementiert werden!
- Beide Annotationsarten dürfen **nicht** gemischt angewandt werden innerhalb einer Entity-Klasse! Best practice ist inzwischen an der get-Methode.



JPA – Anmerkungen zur Annotation (2)

- Mit Hilfe **optionaler(!)** Annotationen können die Default-Bezeichnungen verändert werden oder Eigenschaften von Attributen näher spezifiziert

```
@Entity
@Table(name = "InternetBestellungen")
public class Bestellung ... {
    ...
    @Column(nullable = false)
    private Boolean status;
    ...
    @ManyToOne
    @JoinColumn(nullable = false)
    private Kunde kunde;
    ...
}
```

- Was bedeutet diese Annotation für den Fremdschlüssel in der Datenbank?



JPA – Assoziationen

- Die JPA unterstützt die folgenden Beziehungstypen mit ihren Annotationen:
 - 1:1 mit der Annotation @OneToOne,
 - ✓ 1:n mit der Annotation @OneToMany,
 - ✓ n:1 mit der Annotation @ManyToOne,
 - n:m mit der Annotation @ManyToMany.



JPA – n:m-Beziehungen (1)

- Wie werden n:m-Beziehungen im Relationenmodell umgesetzt?



- Unidirektionale Variante:

```
@Entity
public class Student ... {
    ...
}
...
```

```
@Entity
public class Vorlesung ... {
    ...
    @ManyToMany
    private Collection<Student> studenten = new HashSet();
    public Collection<Student> getStudenten() {
        return studenten;
    }
    ...
}
```



JPA – n:m-Beziehungen (2)

- Bidirektionale Variante:



- Nicht vergessen: Konsistenz der Bidirektionalität muss für neu angelegte bzw. geänderte Objekte in der Anwendung sichergestellt werden!

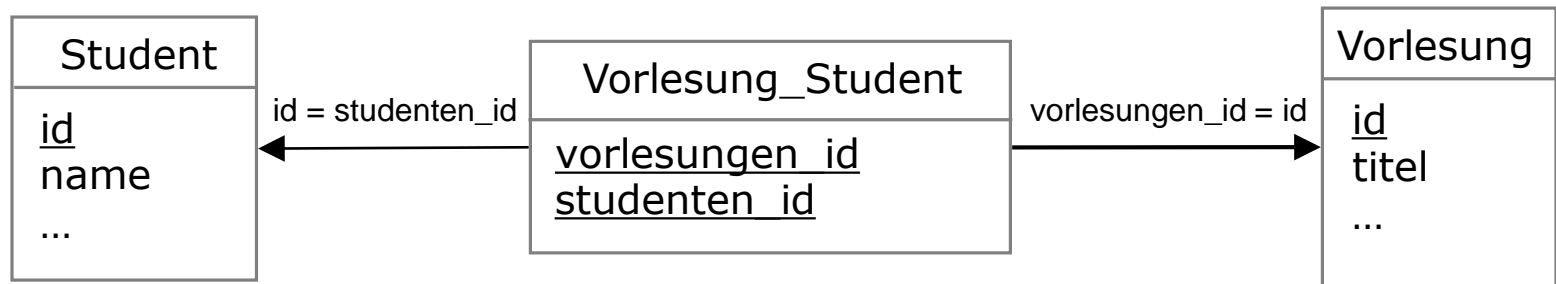
```
@Entity
public class Student ... {
    ...
    @ManyToMany(mappedBy = "studenten")
    private Collection<Vorlesung> vorlesungen =
        new HashSet();
    public Collection<Vorlesung> getVorlesungen() {
        return vorlesungen;
    }
    ...
}
```

```
@Entity
public class Vorlesung ... {
    ...
    @ManyToMany
    private Collection<Student> studenten =
        new HashSet();
    public Collection<Student> getStudenten() {
        return studenten;
    }
    ...
}
```



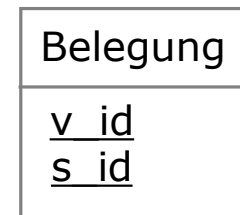
JPA – n:m-Beziehungen (3)

- Ergebnis in der Datenbank



- Der Name der Verbindungstabelle kann (aber muss nicht!) mit Hilfe von Annotations spezifiziert werden, ebenso die Namen der Fremdschlüsselattribute:

```
@Entity
public class Vorlesung ... {
    ...
    @ManyToMany
    @JoinTable (name = "Belegung",
        joinColumns = @JoinColumn(name="v_id"),
        inverseJoinColumns = @JoinColumn(name="s_id")
    )
    private Collection<Vorlesung> vorlesungen = new HashSet();
    ...
}
```





JPA – n:m-Beziehungen (4)

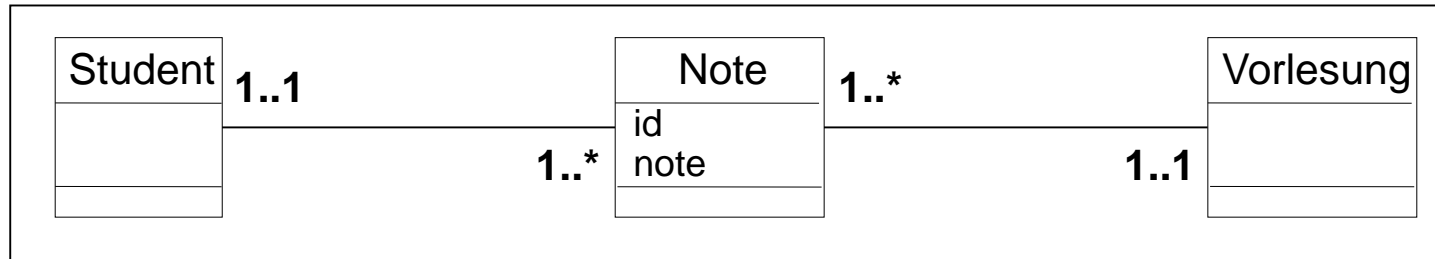
- ACHTUNG: Die Persistence-Provider verhalten sich hier unterschiedlich:
 - Verwendet man in der Anwendung die JPA-API und Hibernate als JPA-Persistence-Provider werden die Primärschlüssel in der Join-Tabelle nicht automatisch erzeugt.
 - Verwendet man stattdessen direkt die Hibernate-API wird ein passender Primärschlüssel erzeugt.
 - Work-around: Verwendung von `@UniqueConstraints`:

```
@Entity
public class Vorlesung ... {
    ...
    @ManyToMany
    @JoinTable (name = "Belegung",
        joinColumns = @JoinColumn(name="v_id"),
        inverseJoinColumns = @JoinColumn(name="s_id")
        uniqueConstraints = @UniqueConstraint(columnNames={v_id, s_id})
    )
    private Collection<Vorlesung> vorlesungen = new HashSet();
    ...
}
```




JPA – n:m-Beziehungen (5)

- m:n-Beziehungen können Beziehungsattribute haben (in unserem Beispiel: Note)
- Lösung: Assoziation muss als eigene Klasse modelliert werden
⇒ zwei 1:n Beziehungen



```
@Entity
public class Note ... {
    @Id
    @GeneratedValue
    private Long id;
    private float note;
    ...
    @ManyToOne
    private Student student;
    @ManyToOne
    private Vorlesung vorlesung;
    ...
}
```

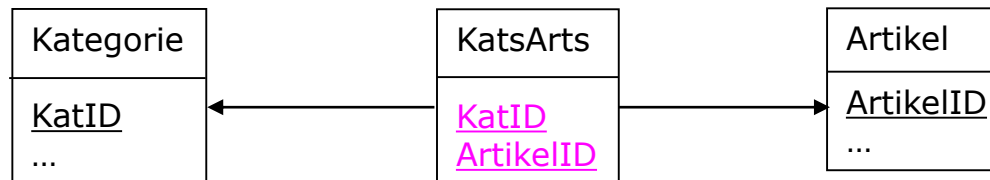


JPA – Bidirektionale n:m-Beziehungen

Nun kann jeder Artikel mehreren Kategorien zugeordnet sein:



→ Welche Seite übernimmt die **Kontrolle für das Update der assoziativen Tabelle KatsArts** bei Änderungen in den beiden Referenzattributen meineArtikel und kategorien vom Typ Collection?



→ Die inverse Seite einer m:n-Beziehung ist frei wählbar, die andere Seite ist dann die Besitzerseite (Owner) und hat **alleine die Kontrolle über die Synchronisation mit der Datenbank** beim Einfügen, Ändern oder Löschen von Beziehungen über ihr Referenzattribut vom Typ Collection!



JPA – Bidirektionale n:m-Beziehungen

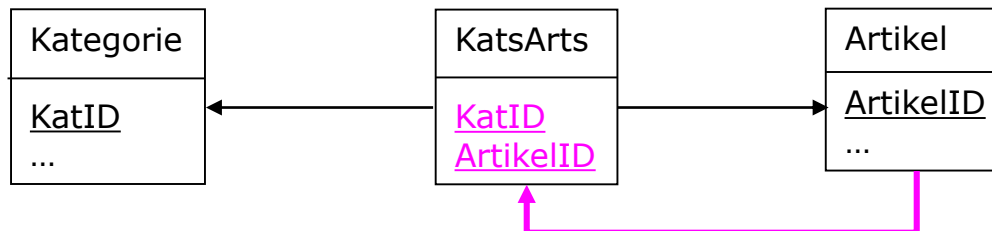
Ein Beispiel: Gegeben Kategorie- und Artikel-Instanzen k1, k2 bzw. a1, a2



```
@Entity
public class Kategorie ... { ...
    private Collection<Artikel> meineArtikel =
        new HashSet();
    @ManyToMany(mappedBy = "kategorien")
    public Collection<Artikel> getMeineArtikel() {
        return meineArtikel;
    }
    ...
}
```

Wert des mappedBy-Attributes = Name
des Referenzattributes in der Besitzer-Klasse.

```
@Entity
public class Artikel ... {
    ...
    private Collection<Kategorie> kategorien;
    @ManyToMany
    public Collection<Kategorie> getKategorien() {
        return kategorien;
    }
    ...
}
```



```
a1.getKategorien().add(k1);
a1.getKategorien().add(k2);
a2.getKategorien().add(k1);
```

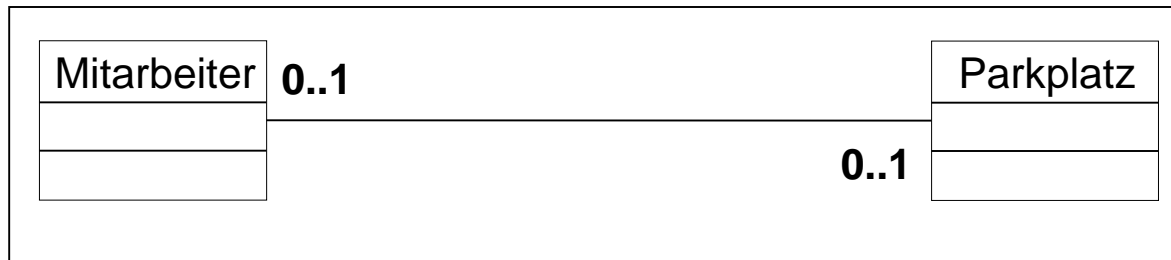
```
k1.getMeineArtikel().add(a2);
k2.getMeineArtikel().add(a1);
k2.getMeineArtikel().add(a2);
```





JPA – 1:1-Beziehungen

- Welche Ausprägungen gibt es für 1:1-Beziehungen?
- Wann sollte eine 1:1-Beziehung als unidirektional und wann als bidirektional implementiert werden?



```
@Entity
public class Mitarbeiter ... {
    ...
}
...
```

```
@Entity
public class Parkplatz ... {
    @OneToOne
    private Mitarbeiter mitarbeiter;

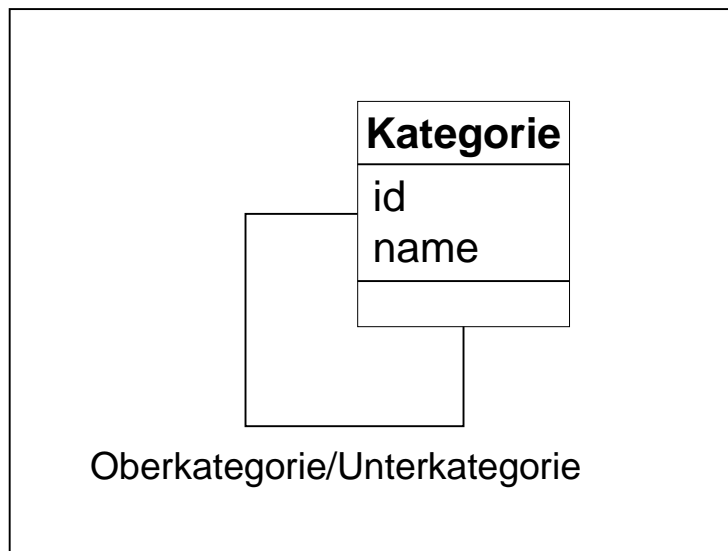
    public Mitarbeiter getMitarbeiter() {
        return mitarbeiter;
    }
    ...
}
```

- Auf welcher Seite wird der Fremdschlüssel erzeugt?



JPA – Reflexive Assoziationen

- Welche Multiplizitäten sind für reflexive Beziehungen möglich?
- Wie erfolgt die OR-Mapping-Umsetzung?





JPA – Collections

- Mehrwertige Assoziationen zwischen Entities werden als Collections innerhalb der Java-Klassen implementiert –
 - die **Collection ist der Datentyp des Referenzattributes**, durch das eine mehrwertige Assoziation implementiert wird.
- Interfaces, die verwendet werden können, um Collections als Typ für Attribute von persistenten Klassen zu deklarieren:
 - java.util.Collection
 - java.util.Set
 - java.util.List
 - java.util.Mapsowie selbst definierte (abgeleitete) Interfaces
- **Persistentes Speichern von Collection-Instanzvariablen** erfolgt per default **nicht** automatisch, wenn sie von einer Entität referenziert werden (gleiches gilt für das Löschen)
 - ⇒ Veränderung des Default-Verhaltens möglich (siehe Abschnitte Persistenz Teil 1 und 2)



JPA – Collections: Ordnung (1)

- Ordnung auf Collections kann auf zwei verschiedene Weisen erzwungen werden: **@OrderBy** vs. **@OrderColumn**
- **@OrderBy**

@Entity

```
public class Konto {  
    ...  
    private BigDecimal kontostand;  
    @OneToMany (mappedBy = "konto")  
    // nach primary key sortiert  
    @OrderBy  
    private List <Buchung> buchungen;  
    ... }  
}
```

@Entity

```
public class Konto {  
    ...  
    private BigDecimal kontostand;  
    @OneToMany (mappedBy = "konto")  
    @OrderBy("betrag")  
    private List <Buchung> buchungen;  
    ... }  
}
```

@Entity

```
public class Buchung {  
    ...  
    private BigDecimal betrag ;  
    @Temporal (TemporalType.DATE )  
    private Date datum;  
    private String text;  
    private Buchungsart buchungsart;  
    @ManyToOne (optional = false)  
    @JoinColumn (name="konto", nullable = false)  
    private Konto konto ;  
    ...  
}
```

Realisierung:

... ORDER BY ... im SQL-Statement



JPA – Collections: Ordnung (2)

- **@OrderColumn**
 - **Persistierung** der Reihenfolge

@Entity

```
public class Konto {  
    ...  
    private BigDecimal kontostand;  
    @OneToMany (mappedBy = "konto")  
    // nach primary key sortiert  
    @OrderColumn (name = "reihenfolge")  
    private List <Buchung> buchungen;  
    ... }  
}
```

@Entity

```
public class Buchung {  
    ...  
    private BigDecimal betrag ;  
    @Temporal (TemporalType.DATE )  
    private Date datum;  
    private String text;  
    private Buchungsart buchungsart;  
    @ManyToOne (optional = false)  
    @JoinColumn (nullable = false)  
    private Konto konto ;  
    ...  
}
```

BUCHUNG:

...	KONTO_ID	REIHENFOLGE
	<NOT NULL>	



JPA – Collections: Map (1)

- Durchlaufen größerer Collections zum Zugriff auf bestimmte Entities ist ineffizient
- Verwendung einer Map!
- Map<Key, Value>
- Key kann sein:
 - Basistyp
 - Embeddee Object (siehe Komposition
 - Entity
- Value kann sein:
 - Basistyp
 - Embedded Object
 - Entity



JPA – Collections: Map (2)

- Maps können auch zur Implementierung von **mehrstelligen M:N-Beziehungen** verwendet werden
- Beispiel: **Kritiker** empfehlen zu einem bestimmten **Gericht** einen bestimmten **Wein**
 - UML-Modellierungsvarianten?
 - Implementierung mit Map:

@Entity

```
public class Kritiker {  
    ...  
    private String name;  
  
    @ManyToMany  
    private Map<Gericht, Wein> empfehlung = new HashMap<>();  
    ...  
}
```

KRITIKER_EMPFEHLUNG:

KRITIKER_ID	GERICHT_ID	WEIN_ID



JPA – Collections: @ElementCollection

- Seit JPA 2.0 können nicht nur Referenzattribute sondern auch Basistypen oder sogar Value-Types (siehe Komposition) innerhalb einer Entität als Collection definiert werden ⇒ **@ElementCollection**

@Entity

```
public class Kunde {
```

@Id

```
    private int id;
```

```
    private String name;
```

@ElementCollection

```
    private Collection<String> nickNames = new HashSet();
```

```
    ...
```

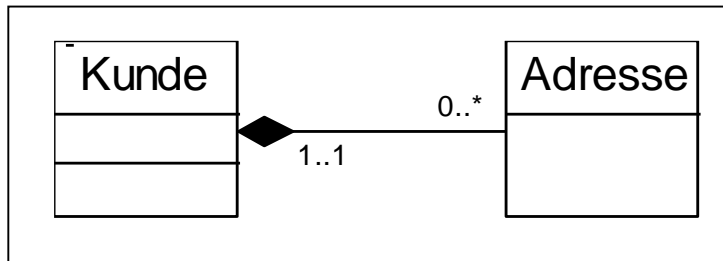
```
}
```

- Was passiert beim Mapping auf die Datenbank?
- Anmerkung 1: Auch hier können wieder die Typen Collection, Set, List oder Map verwendet werden.
- Anmerkung 2: Die @ElementCollection gehört zur Entität und wird damit automatisch mit dieser persistiert bzw. gelöscht.



JPA – Komposition

- **Komponenten** einer Komposition werden bei JPA als **Value-Typen** bezeichnet, die
 - keinen Primärschlüssel besitzen und
 - keinen eigenen Lebenszyklus, im Gegensatz zu Entity-Typen *).



- Die Lebensdauer und ihr Zustand sind immer an die zugehörige Entität gebunden!
- Ein Value-Typ wird als **@Embeddable** annotiert und kann von der zugehörigen Entität durch die Annotation **@ElementCollection** oder **@Embedded** (falls es sich um eine 1:1 Beziehung handelt) als Komponente integriert werden

*) vgl. hier auch die Analogie zur OR-Definition von Objekt-ADT und **Wert-ADT** im SQL3-Standard!



JPA – Komposition: Beispiel

```
import javax.persistence.Embeddable;
...
@Embeddable
public class Adresse {
    private String strasse;
    private String stadt;
    ...
    // get-/set-Methoden
}
```

```
@Entity
public class Kunde {
    @Id
    private int id;
    private String name;
    @ElementCollection
    private List<Adresse> adressen = new ArrayList<Adresse>();
    ...
}
```

- Ergebnis in der Datenbank?
- Um einen Kunden mit seinen Adressen in der Datenbank persistent zu machen, muss nur **em.persist()** für das Kundenobjekt aufgerufen werden.



JPA – Komposition: Spezialfall 1:1

- Falls die Komposition eine 1:1 Beziehung ist, wird das Schlüsselwort `@Embedded` verwendet

```
import javax.persistence.Embeddable;
...
@Embeddable
public class Adresse {
    // Attributliste ..
    // get-/set-Methoden
}
```

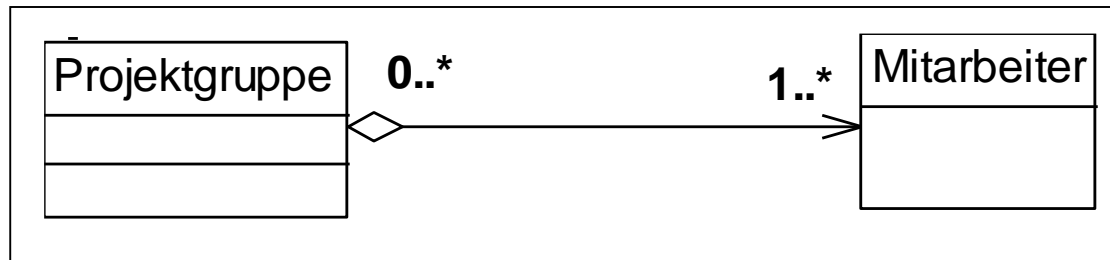
```
@Entity
public class Lieferant {
    @Id
    private int id;
    private String name;
    @Embedded
    private Adresse adresse;
    ...
}
```

- Ergebnis in der Datenbank im Unterschied zum vorherigen Beispiel?

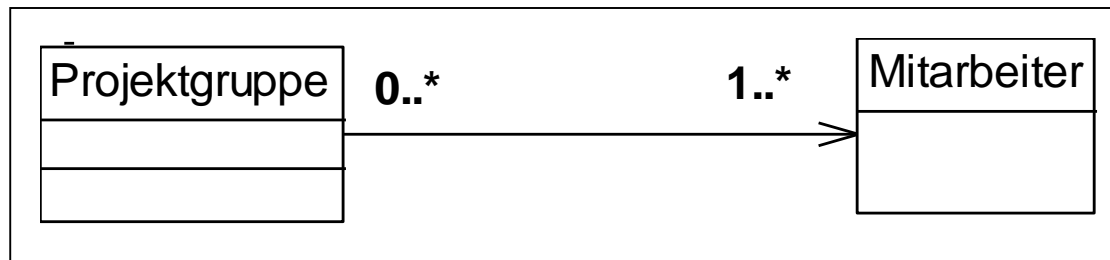


JPA – Aggregation

- Die **Aggregation** stellt im Unterschied zur Komposition eine schwache Beziehung dar. Die Lebensdauer der Komponenten eines Objektes ist nicht an die Lebensdauer des Aggregationsobjektes gebunden!



- Die Komponente kann in diesem Fall mit Hilfe einer geeigneten Assoziation zum Aggregationsobjekt implementiert werden:

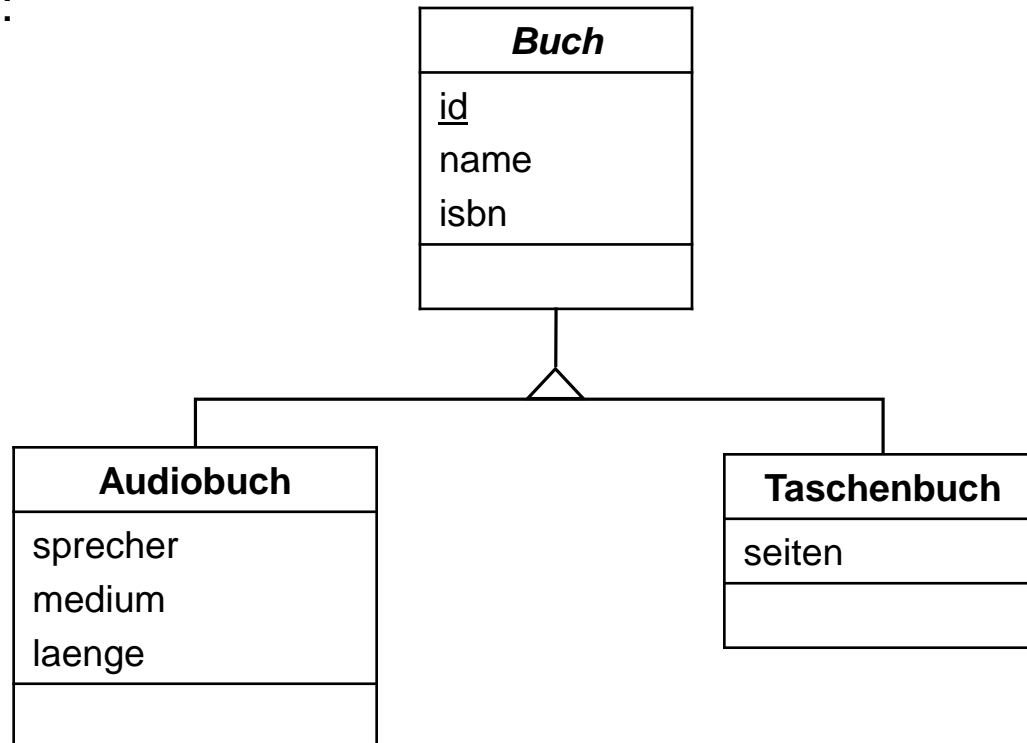




Vererbung

Vorbereitung (Wiederholung)

- Vererbung gibt es auch im (erweiterten) Entity-Relationship-Modell
- Wie kann Vererbung im relationalen Datenmodell umgesetzt werden?
- Beispiel:





JPA – Vererbung: JOINED (1)

- JOINED: Für jede(!) Klasse eine Tabelle

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class Buch ... {
    protected Long id;
    protected String name;
    ...
    @ID
    public Long getID() { return id; }
}
...
```

```
@Entity
public class Audiobuch extends Buch ... {
    private String sprecher;
    ...
}
```



JPA – Vererbung: JOINED (2)

- Ergebnis:

Buch

<u>ID</u>	NAME	ISBN
1	Herr der Ringe Bd. 1	360...
2	Gert Heidenreich	386...

Audiobuch

<u>ID</u>	SPRECHER	MEDIUM	LAENGE
2	Gert Heidenreich	CD	148

Taschenbuch

<u>ID</u>	SEITEN
1	677

- Vorteile?
- Nachteile?
- Wie greift man auf alle Taschenbücher zu?
- Wie greift man auf alle Bücher zu?



JPA – Vererbung: SINGLE_TABLE (1)

- SINGLE_TABLE: Alle Klassen der Vererbungshierarchie werden in einer einzigen Tabelle abgebildet (Default-Strategie)
- Spezifikation eines *DiscriminatorValue*, damit der Tabelleneintrag auf den richtigen Typ abgebildet werden kann.

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
public abstract class Buch ... {
    protected Long id;
    protected String name;
    ...
    @ID
    public Long getID() { return id; }
}
...
```

```
@Entity
@DiscriminatorValue(value = "Audiobuch")
public class Audiobuch extends Buch ... {
    private String sprecher;
    ...
}
```



JPA – Vererbung: SINGLE_TABLE (2)

- Ergebnis:

Buch

<u>ID</u>	DTYPE	NAME	ISBN	...	SPRECHER	MEDIUM	LAENGE	SEITEN
1	TASCHEN- BUCH	Herr der Ringe Bd. 1	360...	...	NULL	NULL	NULL	677
2	AUDIO- BUCH	Der Hobbit	386...	...	Gert Heidenreich	CD	148	NULL

- Vorteile?
- Nachteile?

- *Anmerkung:* DTYPE ist der default-Name (default-Typ: String); Name und Typ können auch anders spezifiziert werden:

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(
    name="buchtyp",
    discriminatorType=DiscriminatorType.STRING
)
public abstract class Buch ... {
    ...
}
```



JPA Vererbung: TABLE_PER_CLASS(1)

- TABLE_PER_CLASS: für jede konkrete Klasse eine Tabelle
- Anmerkung: TABLE_PER_CLASS ist in JPA definiert, ein Persistenzprovider muss diese Strategie aber nicht zwingend implementieren (in Hibernate ist es implementiert)!

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class Buch ... {
    protected Long id;
    protected String name;
    ...
    @ID
    public Long getID() { return id; }
}
...
```

```
@Entity
public class Audiobuch extends Buch ... {
    private String sprecher;
    ...
}
```



JPA Vererbung: TABLE_PER_CLASS (2)

- Ergebnis:

Audiobuch

<u>ID</u>	NAME	ISBN	SEITEN
1	Herr der Ringe Bd. 1	360...	677

Taschenbuch

<u>ID</u>	NAME	ISBN	SPRECHER	MEDIUM	LAENGE
2	Der Hobbit	386...	Gert Heidenreich	CD	148

- Vorteile?
- Nachteile?
- Wie greift man auf alle Bücher zu?
- Was passiert, wenn es eine Beziehung der abstrakten Klasse Buch (z.B. zu Verlag) gibt?



JPA – Vererbung: Zusammenfassung

- **Verschiedene Vor- und Nachteile der einzelnen Implementierungsvarianten für Vererbung:**
 - Entscheidung abhängig vom Anwendungskontext
 - Datenmodell (Assoziationen zu anderen Klassen!)
 - Granularität der Vererbung
 - Programmierungsaufwand
 - Performance-Implikationen (Join-Operationen)
- **Anmerkungen:**
 - Wie müsste man ohne OR-Mapping-Tool mit einer Vererbungshierarchie umgehen?



Datenbanken 2

✓ Einführung

2. Java Persistence API

✓ Persistenz Teil 1

✓ Entity-Klassen

✓ Beziehungen: Assoziationen, Komposition/Aggregation, Vererbung

→ Persistenz Teil 2

3. Datenbankabfragen

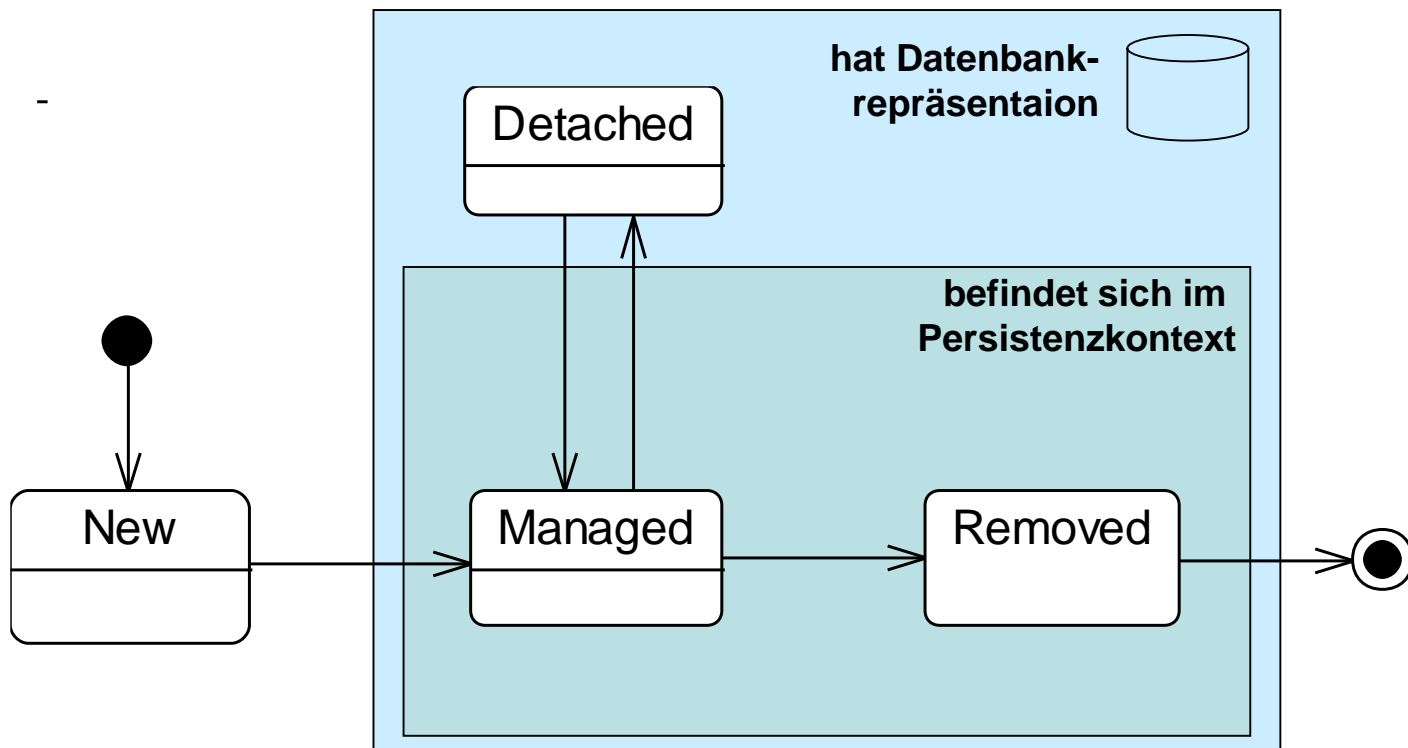
4. Transaktionsmanagement, Caching und Ladestrategien

5. Pufferverwaltung und Optimierung von Zugriffspfaden



JPA – Lebenszyklus einer Entity

- **New** – kein Persistenzkontext, keine DB-Repräsentation, keine Id
- **Managed** – wird in Persistenzkontext verwaltet, DB-Repräsentation
- **Detached** – momentan kein Persistenzkontext, DB-Repräsentation
- **Removed** – Zuordnung zu Persistenzkontext, DB-Repräsentation, zum Löschen freigegeben





JPA – EntityManager Interface (1)

Wichtige Methoden des EntityManager Interface

- public void **persist**(Object entity): übernimmt eine „new“ Entity erstmals bzw. eine bereits verwaltete Entity erneut in den Persistenzkontext
- public <T> T **find**(Class<T> entityClass, Object primaryKey): sucht die Entity im Persistenzkontext, lädt sie ggf. aus der DB mit Hilfe des PK-Wertes
- public <T> T **getReference**(Class<T> entityClass, Object primaryKey): ein Proxy der gesuchten Entity, wird geladen, durch den PK eindeutig identifiziert, übrige Felder können später nachgeladen werden
- public boolean **contains**(Object entity): Gibt an, ob sich die übergebene Entity im Persistenzkontext des EntityManagers befindet.
- public void **refresh**(Object entity): Synchronisation des Zustandes der übergebenen Entity mit dem aktuellen Inhalt der Datenbank (eventuelle Änderungen werden überschrieben)



JPA – EntityManager Interface (2)

Wichtige Methoden der EntityManager Interface (Fortsetzung)

- public void **flush**(): Synchronisation des Zustandes aller Entities, die dem Persistenzkontext zugewiesen sind, mit der Datenbank „in Richtung der DB“
Achtung: flush sollte nur in Ausnahmefällen explizit verwendet werden.
- public void **remove**(Object entity): die Entity wird beim nächsten commit einer Transaktion oder beim nächsten flush() aus der DB gelöscht
- public void **detach**(T entity): übergebene Entity wird aus dem aktuellen Persistenzkontext entfernt, wird dadurch also „detached“
- public <T> T **merge**(T entity): „detached“ Entity wird (als Kopie!) wieder in den Persistenzkontext aufgenommen und dadurch „managed“
- public void **clear**(): Der Persistenzkontext des EntityManagers wird geleert, d.h. alle „managed“ Entities im Kontext werden zu „detached“ Entities, **ohne Synchronisation mit der Datenbank.**
Achtung: clear sollte nur in Ausnahmefällen explizit verwendet werden.



JPA – EntityManager Interface (3)

Zusammenfassende Kategorisierung der vorgestellten Methoden:

- Methoden des Entity Manager, die den CRUD-Operationen auf der Seite des Datenbankservers entsprechen:

CRUD	SQL	EntityManager
Create	INSERT	persist()
Read	SELECT	find()
Update	UPDATE	<i>automatisch</i>
Delete	DELETE	remove()

- Kategorisierung der weiteren Methoden:
 - Löschen von Entities aus dem Persistenzkontext: `detach()`, `clear()`
 - Synchronisation von / zur Datenbank: `refresh()`, `flush()`
 - Lesen mit spezieller Ladestrategie (Laden eines Proxy): `getReference()`
 - Weitere Verwaltungsmethoden für den Persistenzkontext: `contains()`, `merge()`



Entity Manager und Transaktionen

- Wann wird ein Entity Manager geöffnet und wann geschlossen?
 - Was passiert dabei?
- Wann wird eine Transaktion geöffnet und wann geschlossen?
 - Was passiert dabei?
 - Transaktion = „*logical unit of work*“ - Was ist in Anwendungen eine sinnvolle „*logical unit of work*“?
- Eine Variante: Direkte Kopplung von Session und Transaktion

```
EntityManager em = emf.getEntityManager(); // Session wird erzeugt
EntityManager tx = null;
try {
    tx = em.getTransaction();
    tx.begin(); // Start der Transaktion
    // Laden, Verändern, Speichern von Daten ...
    ...
    tx.commit(); // Beenden der Transaktion
}
catch (RuntimeException e) {
    if ( tx != null && tx.isActive() )
        tx.rollback(); // Rollback der Transaktion
    throw e;
}
finally {
    em.close(); // Session wird geschlossen
}
```

- Bei *transaction-scoped persistence context* (nur *container managed*!) endet der persistence context mit Ende der Transaktion, alle Entitäten werden detached.



JPA – Transitive Persistenz

- Das **Attribut cascade** bei einer Beziehungsannotation (@ManyToOne, @OneToMany, @ManyToMany, @OneToOne) bewirkt ein Durchreichen an assoziierte Objekte im Zusammenhang mit dem Aufruf einer persist()-, merge()-, remove()-, refresh()- oder detach()-Methode.

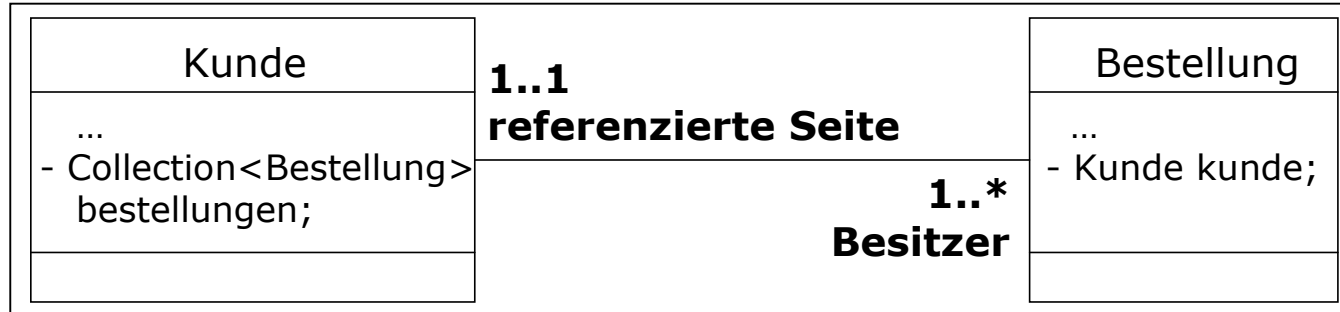
CascadeTypes:

- PERSIST
 - MERGE
 - REMOVE
 - REFRESH
 - DETACH (neu seit JPA 2.0)
 - ALL entspricht cascade = {PERSIST, MERGE, REMOVE, REFRESH, DETACH}
-
- Das **Attribut orphanRemoval** (neu seit JPA 2.0) bei einer Beziehungsannotation @OneToMany oder @OneToOne ermöglicht es, ein nicht mehr über eine Beziehung referenziertes Objekt, also ein „verwaistes“ Objekt, automatisch zu löschen: orphanRemoval="true" – default: "false" (orphan = Waise).



JPA – Transitive Persistenz: Beispiel

Beispiel für den cascade-Type PERSIST



```
@Entity
public class Kunde ... {
    ...
    private Collection<Bestellung> bestellungen =
        new HashSet();

    @OneToMany(mappedBy = "kunde",
        cascade = { CascadeType.PERSIST })
    public Collection<Bestellung> getBestellungen() {
        return bestellungen;
    }
    ...
}
```

```
...
// Transaktion starten
Transaction tx = em.getTransaction().begin();

// persistenten Kunden aus DB holen
Kunde kunde = em.find(Kunde.class, new Long(123));

// nicht-persistente Bestellung zu Kunde hinzufügen
Kunde.addBestellung(bestellung);

// ein expliziter Aufruf von persist(bestellung) ist
// wegen Cascade.Type PERSIST nicht mehr nötig!
tx.commit();
...
```

Kunde → Bestellung speichern



JPA – Callback-Methoden

- Mit Hilfe sogenannter Callback-Methoden können vor (Pre) oder nach (Post) bestimmten Operationen weitere Aktionen ausgeführt werden
- Beispiel (zur Motivation): Mit Hilfe des cascade-Attributs kann das relationale CASCADE auf der Datenbank realisiert werden. Aber was ist mit SET DEFAULT / SET NULL? Was bräuchte man dafür?

⇒ **Callback-Methoden** und ihre Ausführungszeitpunkte

- **@PrePersist** – bevor ein „new“ Entity zum Persistenzkontext hinzugefügt wird
- **@PostPersist** – nachdem ein „new“ Entity in der DB gespeichert wurde (initiiert durch commit oder flush)
- **@PostLoad** – nachdem ein Entity aus der DB geladen wurde
- **@PreUpdate** – wenn ein Entity als verändert vom EntityManager entdeckt wird
- **@PostUpdate** – wenn ein Entity in der DB verändert wird (initiiert durch commit oder flush).
- **@PreRemove** – wenn ein Entity zum Löschen markiert wurde
- **@PostRemove** – nachdem ein Entity in der DB gelöscht wurde (initiiert durch commit oder flush).



JPA – Callback-Methoden: Anwendung

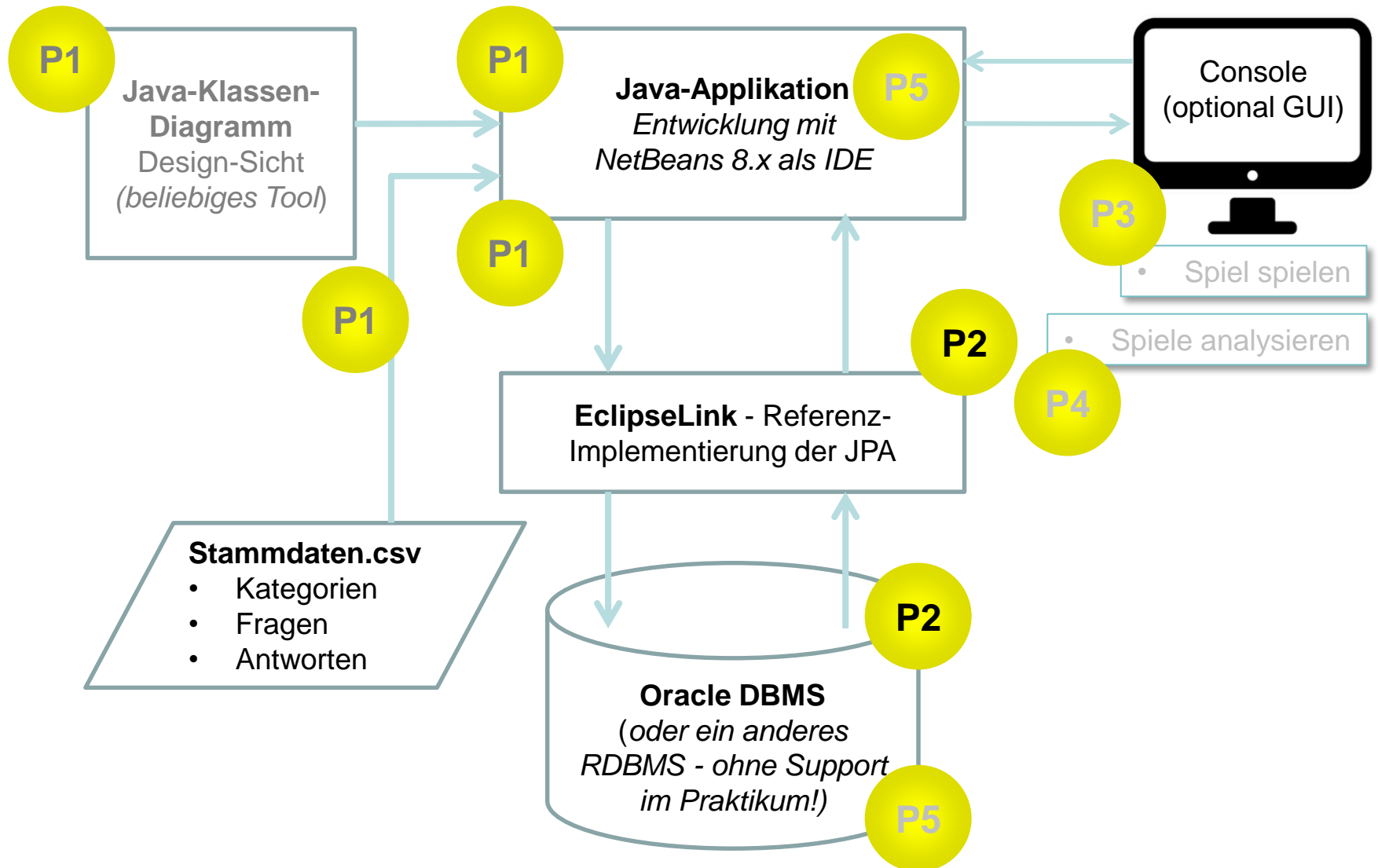
- Callback-Methoden werden in der entsprechenden Entity-Klasse implementiert und durch die jeweilige Annotation gekennzeichnet:

```
@Entity
public class Kunde ... {
    private Int id;
    ...
    private Collection<Bestellung> bestellungen =
        new HashSet();
    ...
    @PostPersist
    void onPostPersist() {
        System.out.println("Kunde " + this.getId() + " wurde in die DB geschrieben");
    }
}
```

- Und wie könnte man SET DEFAULT / SET NULL (siehe vorherige Folie) realisieren?



Praktikum 2





Datenbanken 2

- ✓ Einführung
- ✓ Java Persistence API
 - ✓ Persistenz Teil 1
 - ✓ Entity-Klassen
 - ✓ Beziehungen
 - ✓ Persistenz Teil 2

3. Datenbankabfragen

4. Transaktionsmanagement, Caching und Ladestrategien

5. Pufferverwaltung und Optimierung von Zugriffspfaden