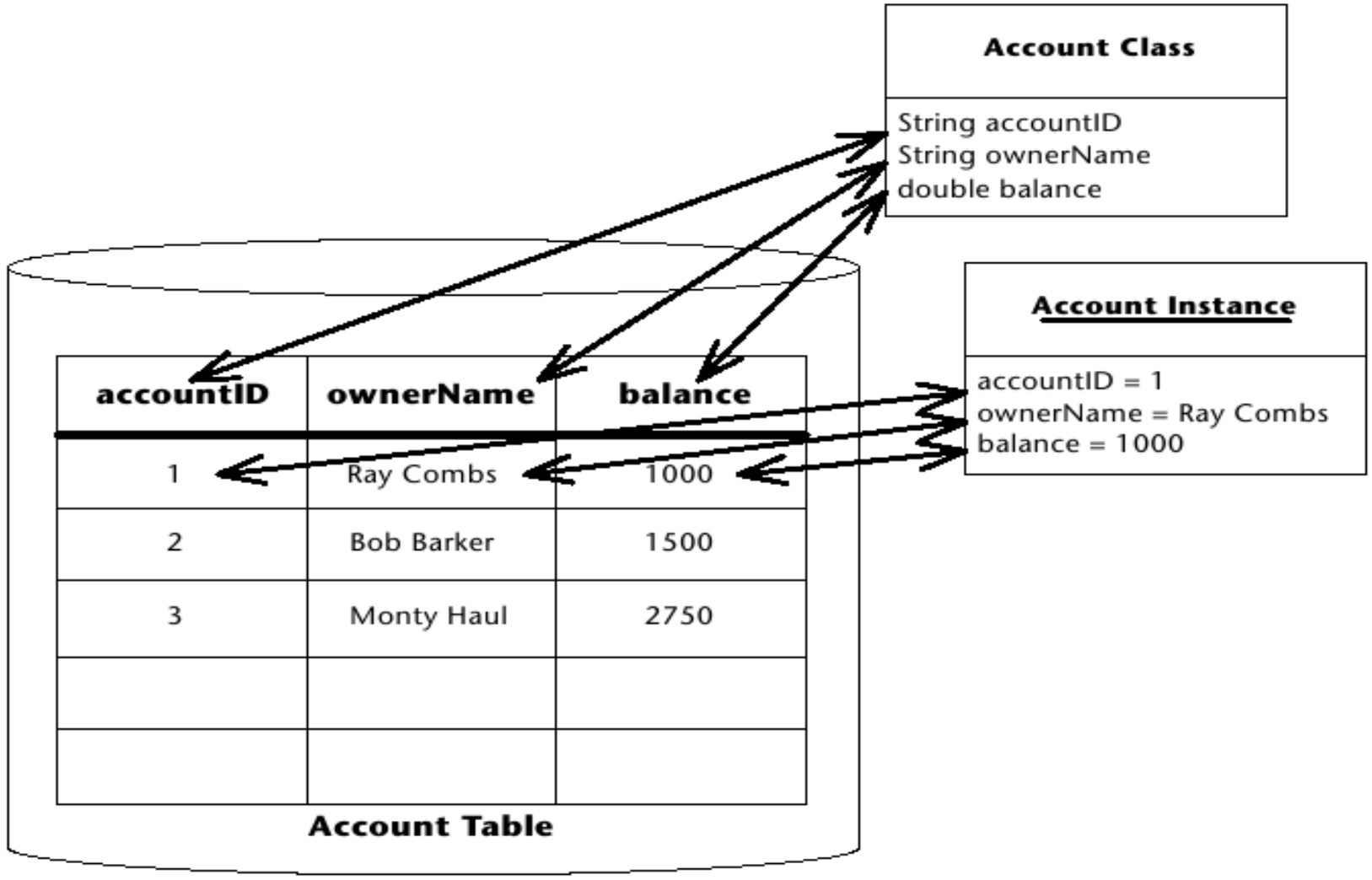
Programmieren mit der Java Enterprise Edition

Java Persistence API

Objekt relationales Mapping

OR-Mapping ist die Abbildung von Objekten auf Datenbanktabellen



Objekt relationales Mapping

- Der einfachste Fall ist jeweils ein Objekt auf eine Tabelle abzubilden.
- Fremdschlüsselbeziehungen auf Datenbank Seite werden auf Klassenseite durch Assoziationen umgesetzt.
- Es gibt 3 Fälle von Fremdschlüsselbeziehungen:
  - 1:1 Beziehung (1 Kunde, 1 Kreditkarte)
  - 1:n Beziehung (1 Kunde, n Bestellungen)
  - n:m Beziehung (n Bestellungen, m Artikel)

Fremdschlüsselbeziehungen

1:1

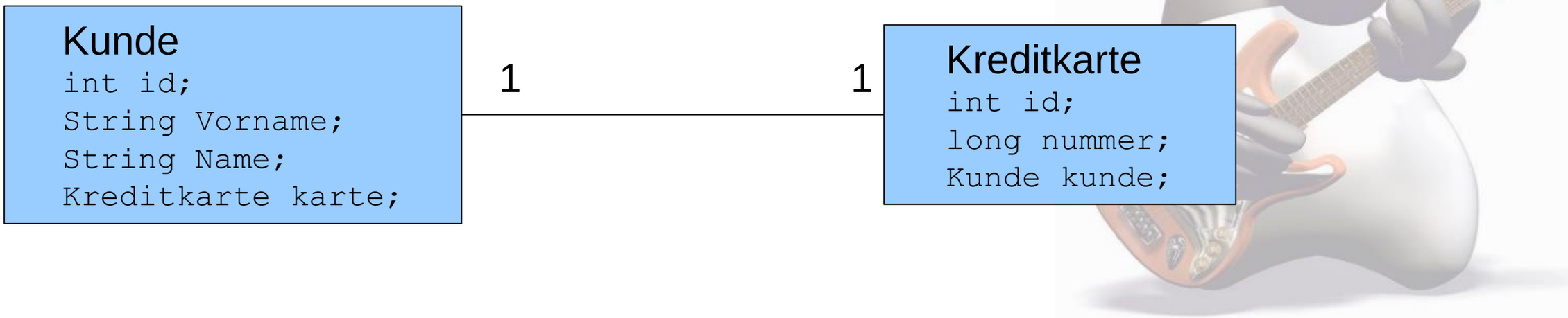
Tabelle Kunde

ID	Vorname	Name	ID_Karte
1	Heinz	Zuber	4
2	Katharina	Horvath	2
3	Martina	Clement	5

Tabelle Kreditkarte

ID	Nummer
4	6543342
2	3232345
5	2345676

- 1:1 Beziehungen werden normalerweise nicht auf 2 Tabellen aufgetrennt
- Eine Tabelle ist ausreichend



Fremdschlüsselbeziehungen

1:n

Tabelle Kunde

ID	Vorname	Name
1	Heinz	Zuber
2	Katharina	Horvath
3	Martina	Clement

Tabelle Bestellung

ID	ID_Kunde
1	1
2	1
3	1
4	2
5	2
6	3

- 1:n Beziehungen werden meistens durch 2 Tabellen ausgedrückt
- Meistens wird in nur einer Tabelle der Primärschlüssel der anderen Tabelle als Fremdschlüssel geführt

ABER:

- 1:n Beziehungen können auch durch eine separate Zwischentabelle definiert werden
- Es kann auch **jeweils** die eine Tabelle den Primärschlüssel der anderen und umgekehrt enthalten (beide Tabellen verweisen aufeinander)

Fremdschlüsselbeziehungen

1:n

Tabelle Kunde

ID	Vorname	Name
1	Heinz	Zuber
2	Katharina	Horvath
3	Martina	Clement

Tabelle Bestellung

ID	ID_Kunde
1	1
2	1
3	1
4	2
5	2
6	3

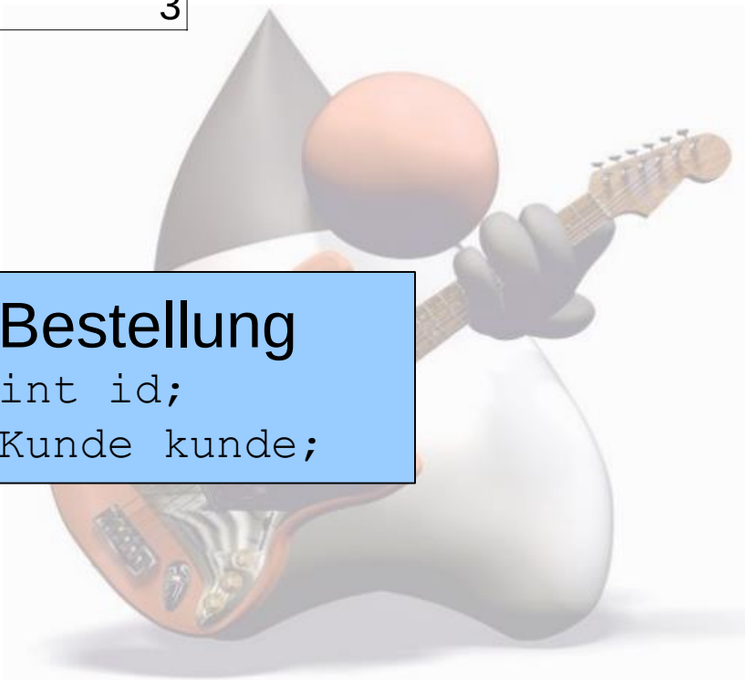
Kunde

```
int id;
String Vorname;
String Name;
Kreditkarte karte;
Bestellung[] bestellungen;
```

1 — n

Bestellung

```
int id;
Kunde kunde;
```



Fremdschlüsselbeziehungen

m:n

Tabelle Bestellung

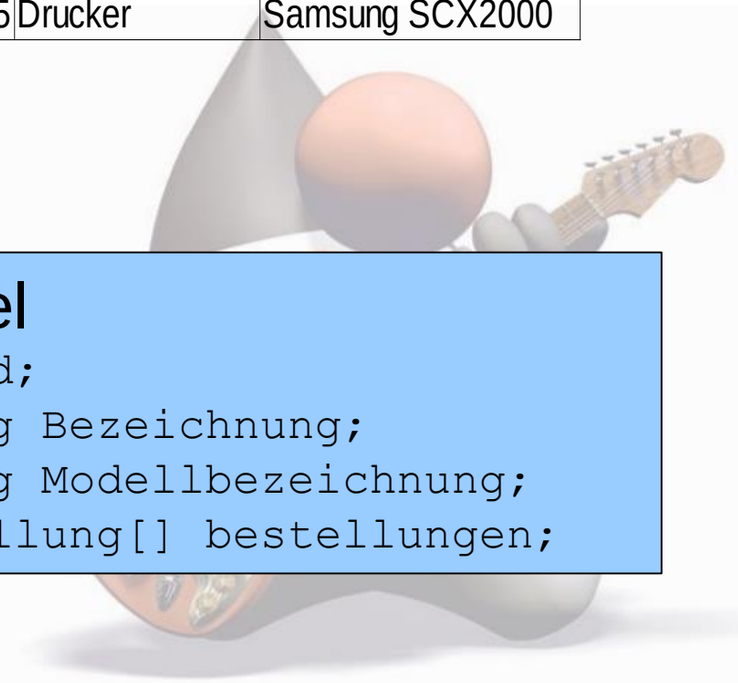
ID	ID_Kunde
1	1
2	1
3	1
4	2
5	2
6	3
7	1

Tabelle Artikel_zu_Bestellung

ID_BESTELLUNG	ID_ARTIKEL
1	3
1	4
1	5
2	4
2	3
2	2

Tabelle Artikel

ID	Bezeichnung	Modellbezeichnung
1	HP Computer	SX 4000
2	Disketten	Verbatim 1.44
3	CD	740 MB
4	Toner	Samsung SCX2000
5	Drucker	Samsung SCX2000



Bestellung

```
int id;
Artikel[] artikelliste;
```

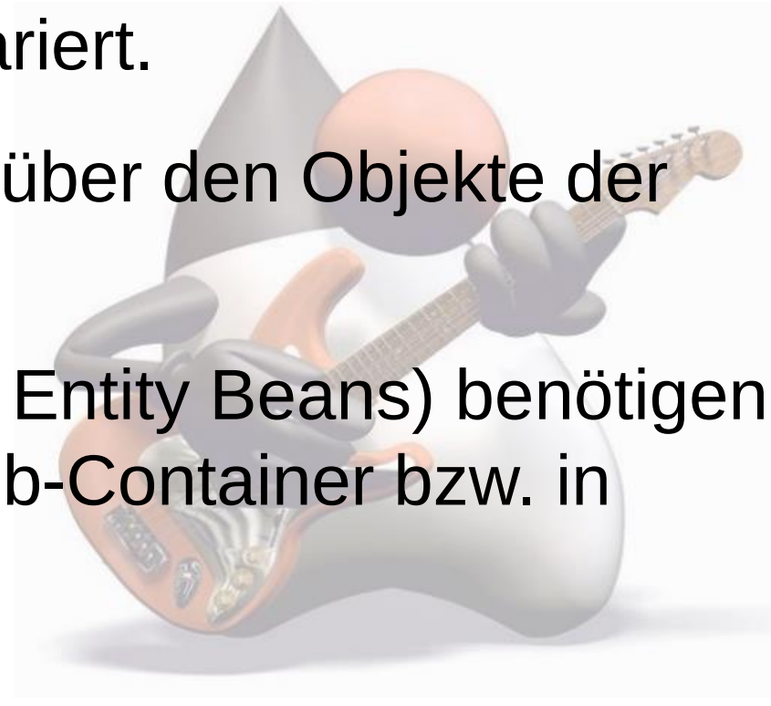
n — m

Artikel

```
int id;
String Bezeichnung;
String Modellbezeichnung;
Bestellung[] bestellungen;
```

Entity Klassen

- Eine Entity Klasse ist eine normale Java Klasse.
  Man sagt auch POJO = Plain Old Java Object
- Zu jeder Entity Klasse gibt es eine Abbildung auf eine (oder mehrere) Tabelle(n) einer relationalen Datenbank. Diese Abbildung auf die Datenbank wird mittels Java Annotations deklariert.
- Jede Entity Klasse hat einen Primärschlüssel, über den Objekte der Klasse eindeutig identifiziert werden können.
- Entity Klassen (im Gegensatz zu den früheren Entity Beans) benötigen keinen EJB Container, sie können auch im Web-Container bzw. in standalone Java Applikationen existieren.



Beispiel Entity Klasse


```
@Entity
@Table(name=”students”)
public class Student implements Serializable {

    @Id
    @Column(name= ”id”)
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name=”matnum”)
    private Long matrikelnummer;

     public Long getId() {return this.id;}
     public void setId(Long id) {this.id = id;}

     public Long getMatrikelnummer() {return matrikelnummer;}
     public void setMatrikelnummer(Long matrikelnummer) {this.matrikelnummer = matrikelnummer;}

}
```

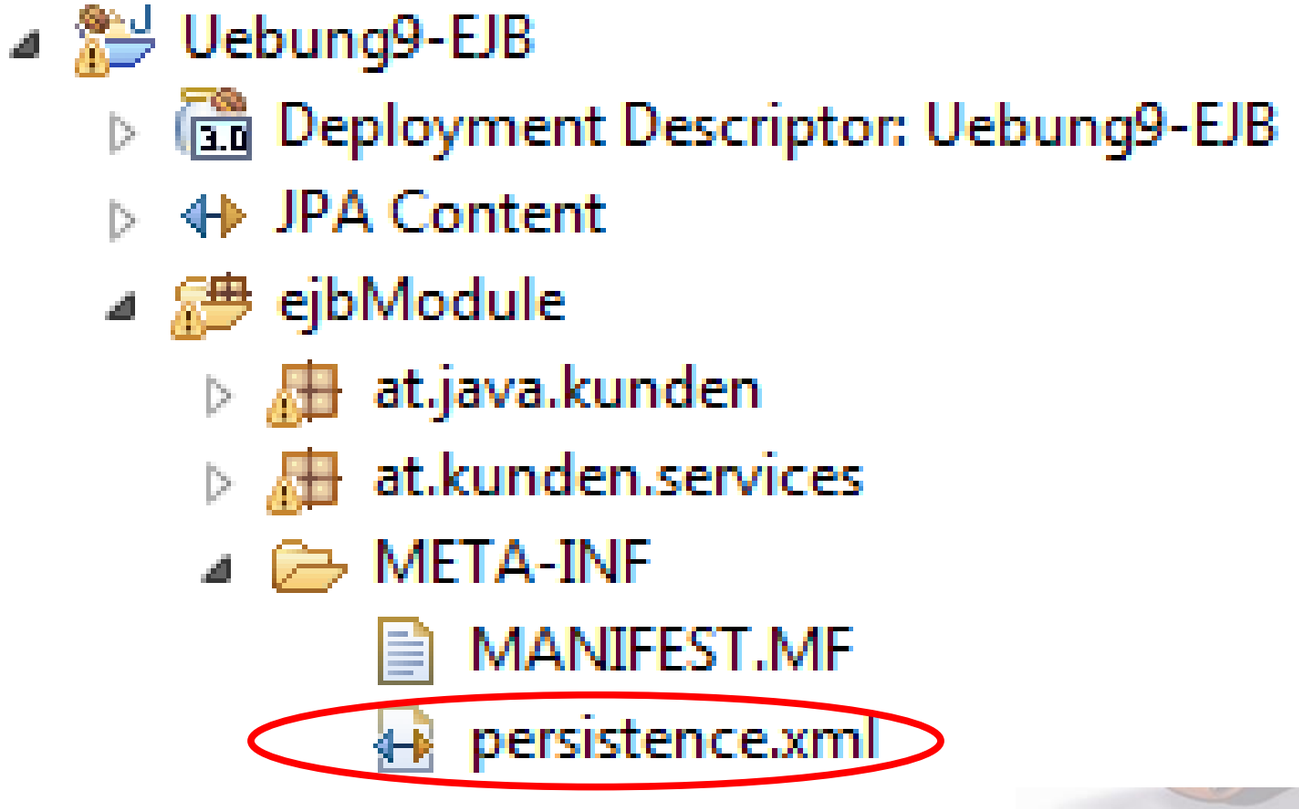
Definition der Abbildung auf die Datenbank

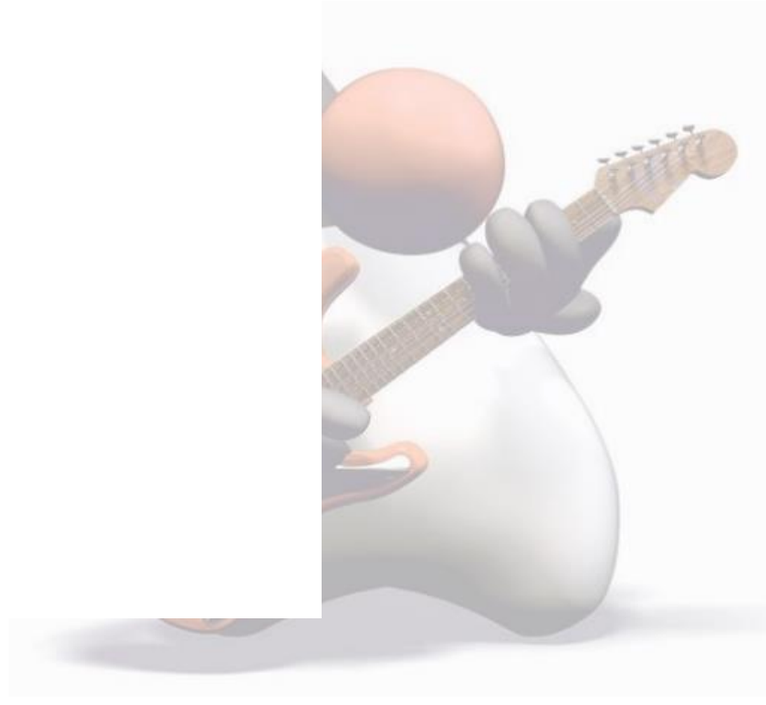
Die Abbildung der Objekte auf die Datenbank erfolgt über Annotations:

@Entity ... definiert die Klasse als Entity Klasse; standardmässig wird Tabellenname = Klassenname angenommen
@Id ... definiert ein Attribut als Primärschlüssel der Entity Klasse
@GeneratedValue(strategy = GenerationType.AUTO) ... sagt dem Persistence Manager, dass der Primärschlüssel der Entity Klasse automatisch generiert werden soll
@Column ... definiert ein zu persistierendes Attribut; standardmäßig wird Attributname = Spaltenname angenommen

Konfiguration der Datenbankverbindung

- Die Datenbankverbindung wir in JavaEE im Applikationsserver konfiguriert und bekommt dort einen Namen(siehe Übung)
- In der Anwendung muss ich die Konfigurationsdatei persistence.xml schreiben.



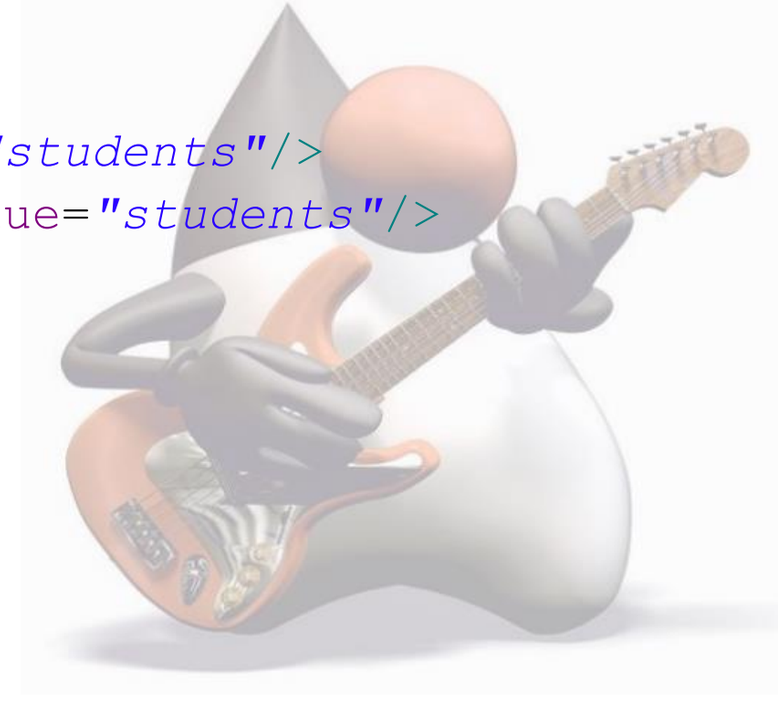


Beispiel persistence.xml Java EE

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence ...>
  <persistence-unit name="Uebung9-EJB" transaction-type="JTA">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <jta-data-source>java:kundendb</jta-data-source>
    <class>at.java.kunden.Kunde</class>
    <exclude-unlisted-classes>false</exclude-unlisted-classes>
    <properties>
        <property name="hibernate.hbm2ddl.auto" value="update" />
    </properties>
  </persistence-unit>
</persistence>
```

Beispiel persistence.xml Java SE

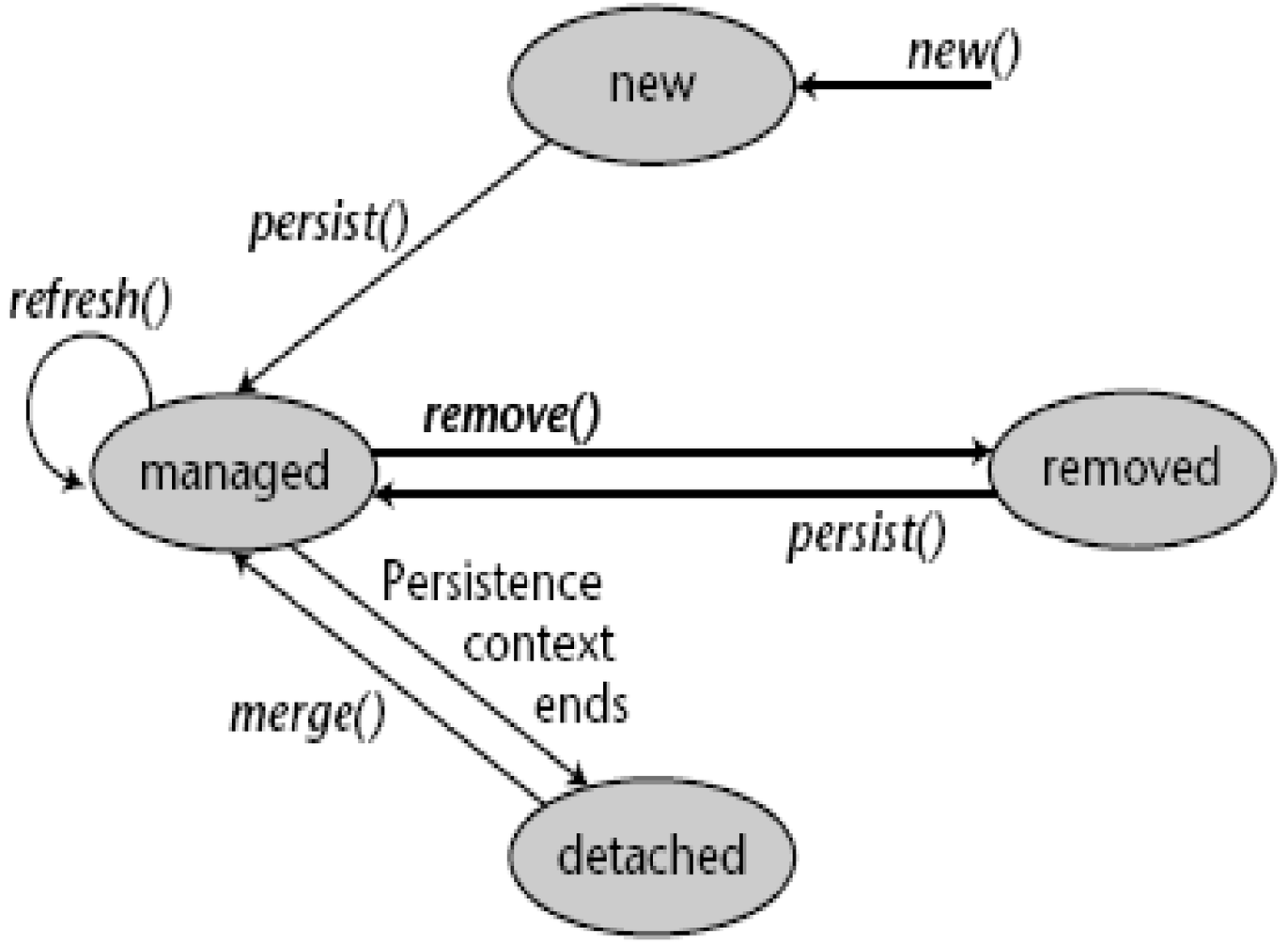
```
<?xml version="1.0" encoding="UTF-8"?>
<persistence …>
  <persistence-unit name= "PU_STUDENTS" transaction-type="RESOURCE_LOCAL">
      <class>de.telekom.Account</class>
      <exclude-unlisted-classes>true</exclude-unlisted-classes>
      <properties>
        <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
        <property name="javax.persistence.jdbc.url"
value="jdbc:mysql://dbhost:3307/students"/>
        <property name="javax.persistence.jdbc.user" value="students"/>
        <property name="javax.persistence.jdbc.password" value="students"/>
      </properties>
  </persistence-unit>
</persistence>
```

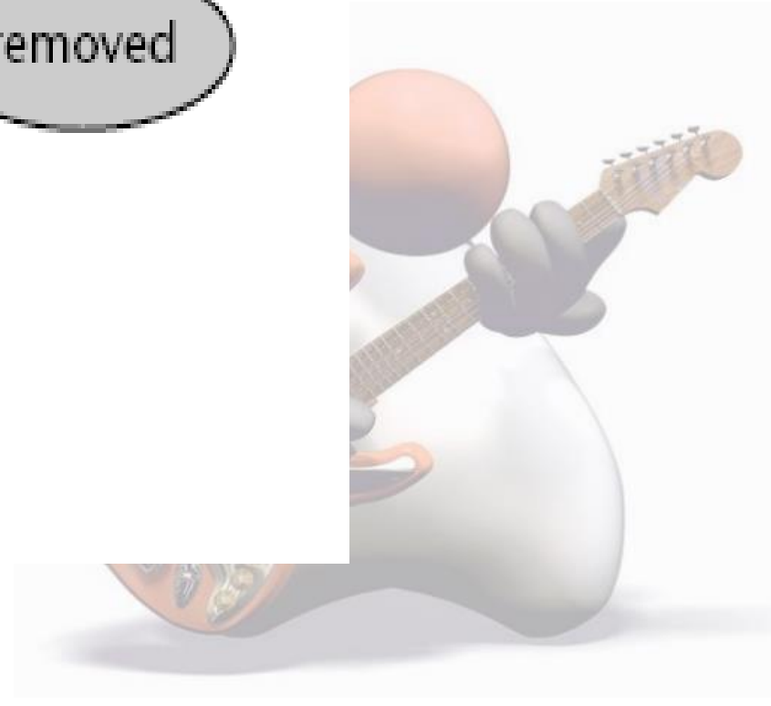


Entity Manager

- Der Entity Manager ist eine Instanz, die den Lebenszyklus der Entity Klassen steuert.
- Er instanziiert Entity Klassen über Suchabfragen gegen die Datenbank
- Er synchronisiert Entity Objekte mit der Datenbank

Lifecycle einer Entity Klasse





Lifecycle eines Entity Objekts

- **New:** Das Objekt wurde mittels *new* Operator in der Java Virtual Machine (JVM) angelegt. Die Objekt hat noch kein Äquivalent in der relationalen Datenbank, es wird noch nicht vom Entity Manager verwaltet.
- **Managed:** Das Objekt wird vom Entity Manager verwaltet. Der Entity Manager weiss, ob Attribute eines Objektes geändert wurden. Er entscheidet, wann geänderte Attribute in die Datenbank geschrieben werden und wann das Objekt neu aus der Datenbank gelesen wird. Das Schreiben in die Datenbank und der Refresh können aber auch programmatisch erzwungen werden.
- **Removed:** Das Objekt existiert noch in der JVM, es wird vom Entity Manager verwaltet, ist aber zur Löschung in der Datenbank vorgemerkt.
- **Detached:** Das Objekt existiert in der Datenbank und in der JVM, es wird aber nicht mehr vom Entity Manager verwaltet (kommt vor, wenn das Objekt zwischen JVMs transportiert wird, oder die Transaktion beendet wurde).

Methoden des Entity Managers

- **persist()**: Ein Objekt im Zustand *new* wird unter die Verwaltung des Persitence-Manager (managed) gestellt und in die Datenbank geschrieben
- **refresh()**: Ein Objekt, das sich im Zustand *managed* befindet, wird erneut aus der Datenbank gelesen
- **remove()**: Ein Objekt, das sich im Zustand *managed* befindet, zum Löschen markiert und mit Beendigung der Transaktion aus der Datenbank gelöscht. Das Java Objekt bleibt bestehen.
- **merge()**: Ein Objekt, das nicht unter der Verwaltung des Persistence Managers steht, das aber unter dem Primärschlüssel schon in der Datenbank steht, wird unter Verwaltung des Persistence Managers gestellt. Die Daten des Objektes werden damit mit Beendigung der Transaktion in die Datenbank geschrieben.
- **flush()**: Alle Änderungen im Objekt werden in die Datenbank geschrieben
- **find()**: Es wird das Objekt mit dem angegebenen Primärschlüssel aus der Datenbank geholt.
- **createQuery**(„select ...“): Eine benutzerdefinierte Datenbankabfrage wird definiert, mittels query.getResultList wird das Ergebnis in Form von Java Objekten ausgelesen.

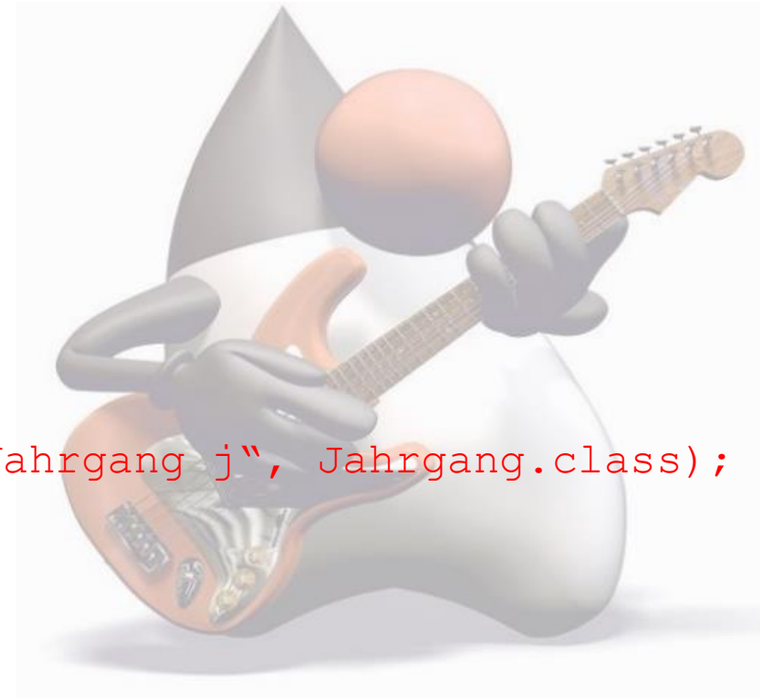
Methoden des Entity Managers

```
@Stateless
@WebService
public class StudentServiceBean implements StudentServiceRemote, StudentServiceLocal {
    @PersistenceContext
    private EntityManager em;

    /** Creates a new instance of StudentServiceBean */
    public StudentServiceBean() {
    }

    public Student getStudentByID(Long id) {
        Student student = em.find(Student.class,id);
        return student;
    }

    public List<Jahrgang> getAllJahrgaenge(){
        TypedQuery<Jahrgang> query = em.createQuery("select j from Jahrgang j", Jahrgang.class);
        List<Jahrgang> jahrganglist = query.getResultList();
        return jahrganglist;
    }
}
```



Methoden des Entity Managers

```
    public void insertStudent(Student student){
         em.persist(student);
    }

    public Student updateStudent(Student student) {
        student = em.merge(student);
        return student;
    }

    public void deleteStudent(Long id){
        Student student = em.find(Student.class, id);
        if (student!=null) em.remove(student);
    }
}
```

```
Adresse adresse = new Adresse();
Jahrgang jahrgang = new Jahrgang();

jahrgang.setJahr(2004);
em.persist(jahrgang);

adresse.setStrasse("Thomas Edison Straße");
adresse.setHausnummer("2");
adresse.setOrt("Eisenstadt");
adresse.setPlz("7000");
em.persist(adresse);

Student student1 = new Student();
student1.setVorname("Peter");
student1.setNachname("Wind");
student1.setMatrikelnummer(123456L);
student1.setEmail("peter.wind@abc.com");
student1.setAdresse(adresse);
student1.setJahrgang(jahrgang);
em.persist(student1);
```

Lifecycle Callback Methoden

Wie bei Session Beans werden die Callbackmethoden mit Annotations gekennzeichnet:

- @PrePersist, @PostPersist
- @PreRemove, @PostRemove
- @PreUpdate, @PostUpdate
- @PostLoad

Fremdschlüsselbeziehungen

Bei der Definition von Fremdschlüssel-beziehungen muss immer bedacht werden, auf welcher Seite der beiden zueinander in Beziehung stehenden Tabellen sich der Fremdschlüssel befindet (siehe nächste Folie):

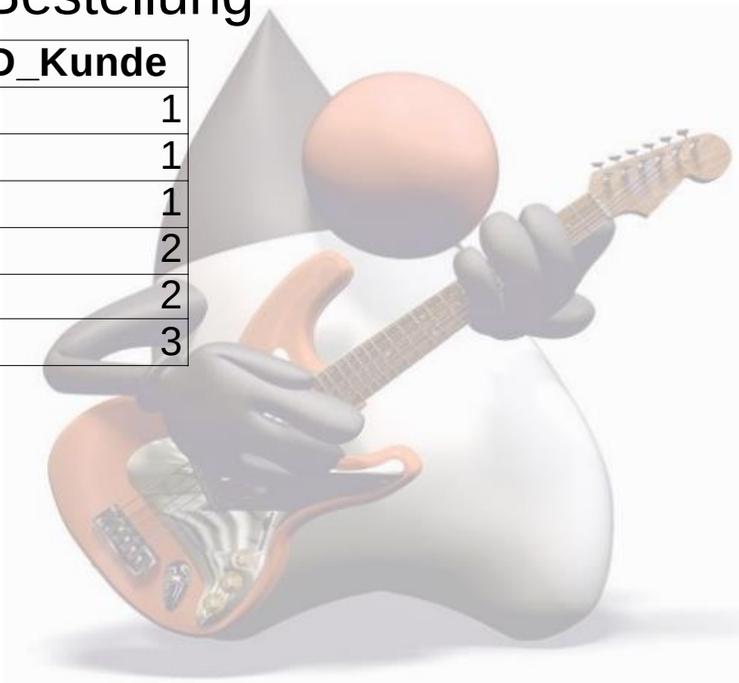
1:n

Tabelle Kunde

ID	Vorname	Name
1	Heinz	Zuber
2	Katharina	Horvath
3	Martina	Clement

Tabelle Bestellung

ID	ID_Kunde
1	1
2	1
3	1
4	2
5	2
6	3



Fremdschlüsselbeziehungen

Die wichtigsten Annotations für die Beziehung von Fremdschlüsselbeziehungen sind:

- @JoinColumn ... definiert ein Attribut als Fremdschlüssel
- @OneToOne ... definiert eine 1:1 Beziehung
- @OneToMany, @ManyToOne ... definiert eine 1:n Beziehung
- @ManyToMany ... definiert eine m:n Beziehung

Fremdschlüsselbeziehungen

1:1

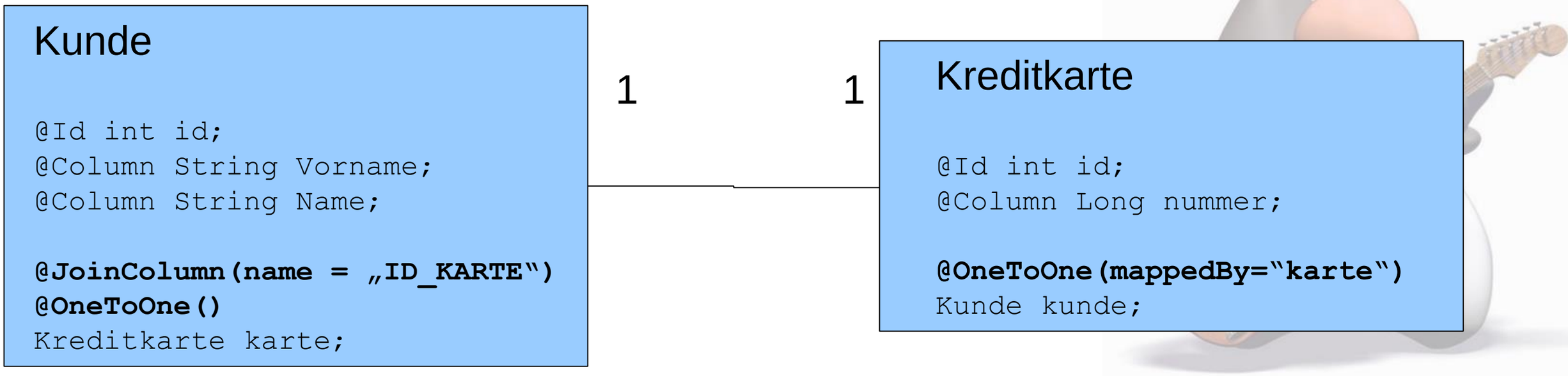
Tabelle Kunde

ID	Vorname	Name	ID_KARTE
1	Heinz	Zuber	5
2	Katharina	Horvath	2
3	Martina	Clement	4

Tabelle Kreditkarte

ID	Nummer
4	6543342
2	3232345
5	2345676

- 1:1 Beziehungen werden normalerweise nicht auf 2 Tabellen aufgetrennt
- Eine Tabelle ist ausreichend



Fremdschlüsselbeziehungen

1:n

Tabelle Kunde

ID	Vorname	Name
1	Heinz	Zuber
2	Katharina	Horvath
3	Martina	Clement

Tabelle Bestellung

ID	ID_Kunde
1	1
2	1
3	1
4	2
5	2
6	3

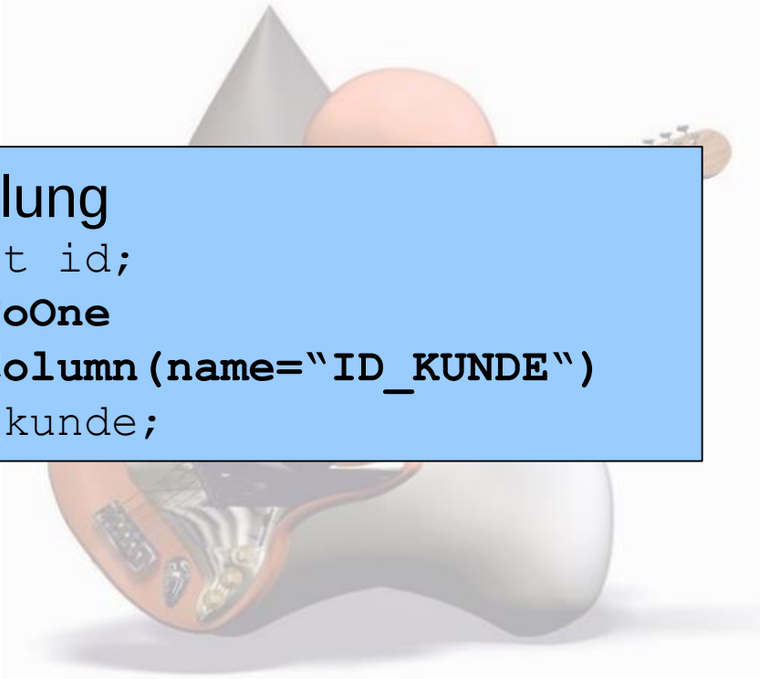
Kunde

```
@Id int id;
@Column String Vorname;
@Column String Name;
@OneToMany(mappedBy="kunde")
List<Bestellung> bestellungen;
```

1 n

Bestellung

```
@Id int id;
@ManyToOne
@JoinColumn(name="ID_KUNDE")
Kunde kunde;
```



m:n

Tabelle Bestellung

ID	ID_Kunde
1	1
2	1
3	1
4	2
5	2
6	3
7	1

Tabelle Artikel_zu_Bestellung

ID_BESTELLUNG	ID_ARTIKEL
1	3
1	4
1	5
2	4
2	3

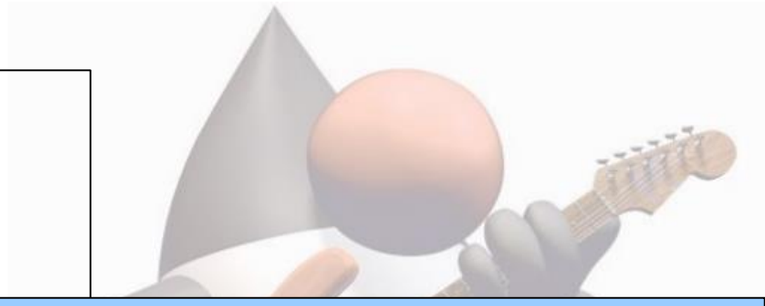
Tabelle Artikel

ID	Bezeichnung	Modellbezeichnung
1	HP Computer	SX 4000
2	Disketten	Verbatim 1.44
3	CD	740 MB
4	Toner	Samsung SCX2000
5	Drucker	Samsung SCX2000

Bestellung

```
@Id int id;
@ManyToMany()
@JoinTable(name="ARTIKEL_ZU_BESTELLUNG",
 joinColumns={@JoinColumn(name="ID_BESTELLUNG")},
 inverseJoinColumns={@JoinColumn(name="ID_ARTIKEL")})
List<Artikel> artikelliste;
```

n



Artikel

m

```
@int id;
@Column String Bezeichnung;
@Column String Modellbezeichnung;
@ManyToMany(mappedBy="artikelliste")
Collection <Bestellung> bestellungen;
```

EJB Query Language

Definition von Queries erfolgt vorzugsweise in der Entity Klasse

```
@Entity
@NamedQueries({
    @NamedQuery(name="findAllJahrgaenge", query="SELECT j FROM Jahrgang j"),
    @NamedQuery(name="findJahrgangByJahr", query = "SELECT j FROM Jahrgang j WHERE j.jahr=:jahr")
})
public class Jahrgang implements Serializable {

    public Jahrgang() {
    }

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @Column()
    private Integer jahr;
```

EJB Query Language

Aufruf der Query

```
public Jahrgang[] getAllJahrgaenge(){
    Query query = em.createNamedQuery("findAllJahrgaenge");
    List<Jahrgang> jahrganglist = query.getResultList();
    return jahrganglist.toArray(new Jahrgang[10]);
}

public Student[] getStudentsForJahrgang(String strJahr){
    Integer jahr = Integer.parseInt(strJahr);

    Query query = em.createNamedQuery("findStudentByJahr");
    query.setParameter("jahr", jahr);
    List<Student> studenten = query.getResultList();
    return studenten.toArray(new Student[studenten.size()]);
}
```

EJB Query Language

SELECT o FROM Student AS o
ist gleich wie
SELECT o FROM Student o
… holt alle Einträge einer Tabelle

Ausführen einer named Query:

```
em.createNamedQuery(„findAllStudenten“);
List<Student> studenten= query.getResultList();
```

Ausführen einer dynamischen Query:

```
em.createQuery(„SELECT o FROM Student o“);
List<Student> studenten= query.getResultList();
```

wenn ich nur einen Wert auslesen will:

```
Student student = query.getSingleResult();
```

EJB Query Language

Setzen von Parametern:

```
TyperQuery<Student> query = em.createQuery(„SELECT o FROM
Student o WHERE o.alter >= :min_alter AND o.alter <=
:max_alter“, Student.class);
query.setParameter(„min_alter“, 17);
query.setParameter(„max_alter“,46);
List<Student> studenten=query.getResultList();
```

Auslesen einzelner Spalten:

```
Query query = em.createQuery(„SELECT o.vorname, o.nachname
FROM STUDENT o“);
List namen =query.getResultList();
for (Object[] objects: namen){
      String vorname = (String)objects[0];
      String nachname = (String)objects[1];
}
```

EJB Query Language

Navigieren durch Objekte:
SELECT s.jahrgang.jahr FROM Student c

Selektieren von Collections
SELECT r FROM Customer c, IN (c.reservations) r
where r.date = :date

DISTINCT
SELECT DISTINCT s.vorname FROM Student s

NULL values
SELECT s FROM Student s WHERE s.vorname IS NULL

JOINS
SELECT s.name, l.name FROM Student s LEFT JOIN s.lehrer l

EJB Query Language

JOINS in SQL:

```
SELECT A.EineSpalte, B.EineAndereSpalte
FROM Tabelle1 AS A JOIN Tabelle2 AS B
ON A.EinWert = B.EinAndererWert;
```

JOINS IN EJB-QL

```
SELECT s.name, l.name FROM Student s LEFT JOIN s.lehrer l
```

Sortierungen

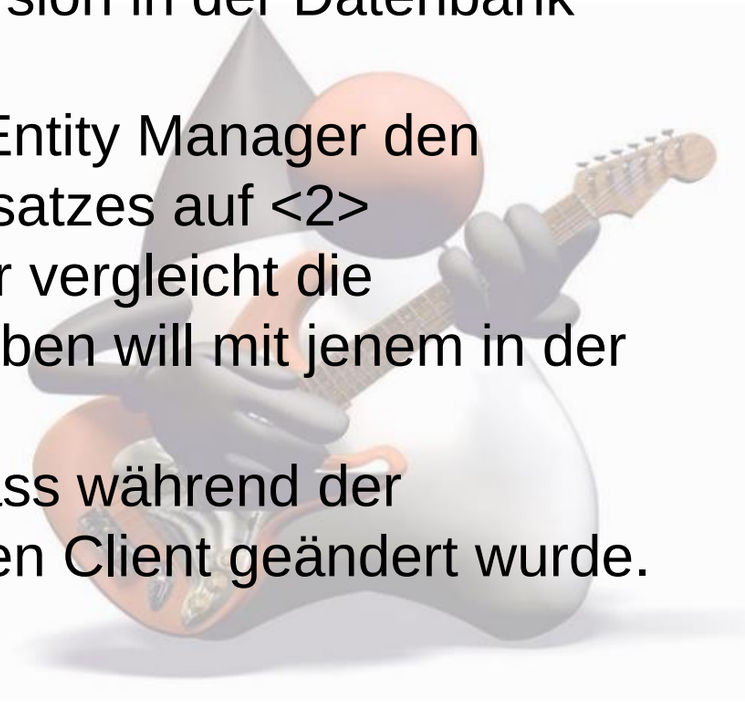
```
SELECT s FROM Student s ORDER BY s.name DESC
SELECT s FROM Student s ORDER BY s.name ASC
```

Optimistische vs. Pessimistische Transaktionslogik

Optimistische Transaktionslogik:

Jede Zeile in einer Datenbanktabelle bekommt eine Versionsnummer. Die Logik läuft dann so ab:

1. Client1 liest einen Datensatz mit der Versionsnummer <1>
2. Client2 liest den gleichen Datensatz mit der Versionsnummer <1>
3. Der Client2 will den Datensatz schreiben. Der Entity Manager vergleicht die Version des Datensatzes den Client2 (<1>) schreiben will mit der Version in der Datenbank (auch <1>)
4. Die Versionsnummern sind gleich, deswegen screibt der Entity Manager den Datensatz und erhöht dabei die Versionsnummer des Datensatzes auf <2>
5. Client1 will den Datensatz schreiben. Der Entity Mananger vergleicht die Versionsnummern des Datensatzes den Client1 (<1>) schreiben will mit jenem in der Datenbank (Version <2>)
6. Der Entity Manager wirft eine Exception, da er erkennt, dass während der Bearbeitung durch Client1 der Datensatz durch einen anderen Client geändert wurde.



Optimistische vs. Pessimistische Transaktionslogik

Pessimistische Transaktionslogik:

Sobald ein Client einen Datensatz zur Bearbeitung holt, wird der entsprechende Datensatz gesperrt:

1. Client1 liest den Datensatz und sperrt ihn gleichzeitig zur Bearbeitung. OK
2. Client2 liest den Datensatz. OK
3. Client2 versucht den Datensatz erneut zu lesen und gleichzeitig zu sperren. Der Entity Manager wirft eine Exception
4. Client1 schreibt den Datensatz und gibt ihn wieder frei.

Optimistische vs. Pessimistische Transaktionslogik

Umsetzung optimistische Transaktionslogik

Die Enity Klasse bekommt ein Attribut das die Versionsnummer führt. Dieses Attribut wird mit der @Version Annotation gekennzeichnet.

```
@Version
private Long version;
public Long getVersion() {return version;}
public void setVersion(Long version) {this.version = version;}
```

Der Entity Manager wirft dann im Konfliktfall eine OptimisticLockException.

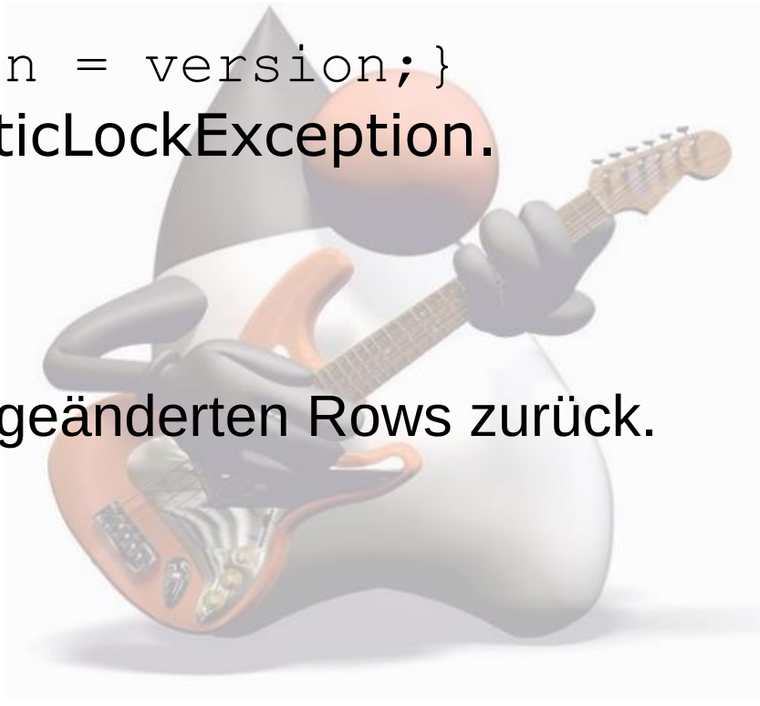
Wie macht das der Entity Manager?

UPDATE student set vorname="Michael", version = version +1

WHERE id = 12 AND version = 101;

Der Entity Manager bekommt von der Datendbank die Anzahl der geänderten Rows zurück.

Wenn diese =0 ist, dann hat es einen Konfilkt gegeben.



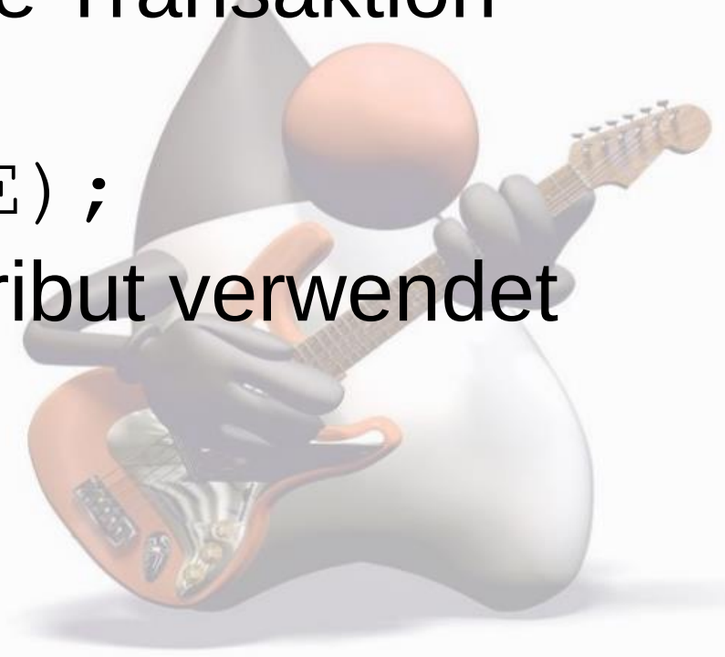
Optimistische vs. Pessimistische Transaktionslogik

Umsetzung pessimistische Transaktionslogik

Mittels einer Lock Methode wird der Datensatz für konkurrierendes Schreiben gesperrt. Während eine Transaktion einen Datensatz gelockt hat, kann keine andere Transaktion schreiben.

```
em.lock(student, LockModeType.WRITE);
```

Auch bei Locking sollte immer ein Versions Attribut verwendet werden.



Kaskadierung

Kaskadierung bedeutet, dass eine Datenbankoperation auch auf die assoziierten Objekte eines Entity Objektes durchgeführt werden.
Beispiel: Ein Student wird in der JVM neu angelegt, ein Adressobjekt wird in der JVM neu angelegt. Das Adressobjekt wird dem Student angehängt. Dann wird persist für das Objekt Student aufgerufen, woraufhin gleichzeitig der Student als auch die Adresse in der Datenbank angelegt werden.
Danach wird der Student aus der Datenbank gelöscht und gleichzeitig wird auch die angehängte Adresse gelöscht.
Kasadierung kann in den Annotations **@OneToMany**, **@ManyToOne**, **@OneToOne** und **@ManyToMany** eingestellt werden.

```
@ManyToMany(cascade=CascadeType.ALL)
@OneToOne(cascade={CascadeType.PERSIST, CascadeType.MERGE})
@OneToMany(cascade={CascadeType.DELETE})
```

Es gibt die Optionen: ALL, PERSIST, MERGE, REMOVE, REFRESH

Contraints

Constraints, also zwingende Bedingungen, können teilweise über Kaskadierungen gelöst werden.
Andererseits können Constraints auf Attributen gesetzt werden:
@Column(nullable=false,unique=true)

Es kann gesteuert werden, ob Attribute geschrieben werden dürfen:
@Column(insertable=false,updatable=false)

Diese Attribute können auch für @JoinColumn gesetzt werden.

Lazy Loading

Über Lazy Loading kann gesteuert werden, ob assoziierte Objekte sofort aus der Datenbank geladen werden sollen oder erst dann, wenn darauf zugegriffen wird.

@OneToMany(fetch=FetchType.EAGER)

... die assoziierten Objekte werden sofort geladen. (Das kann im schlechtesten Fall die ganze Datenbank sein!!!)

@ManyToMany(fetch=FetchType.LAZY)

... dem Entity Manager wird vorgesschlagen, die Attribute erst dann zu Laden, wenn auf sie zugegriffen wird.

Programmieren mit der Java Enterprise Edition

Ende
Java Persistence API