Datenbanksysteme 2

Java Persistence

- Java Platform
- JEE (Jakarta EE)
- Jakarta Persistence API

Jakarta Persistence API

- Grundlagen
- Abbildung Klasse, Klassenhierarchien
- Abbildung von Objektbeziehungen
- Transaktionen
- Java Persistence Query Language (JPQL)

Java Persistence

- Grundlagen

Was ist JPA (Jakarta Persistence API)

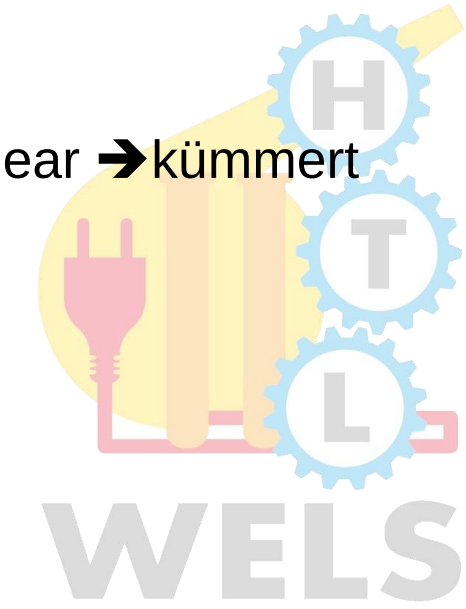
- **Persistenzmechanismus** für Java-Objekte
- interoperabel mit **J2EE, Jakarta EE**, Bestandteil von **EJB 3.0**, **EJB 3.1, EJB 3.2**
- **JPA** ist nur eine **Spezifikation** von SUN/Oracle, JSR 220 (2006), JSR 317 (sagt nichts über die verwendeten Technologien, wie etwa Byte Code Enhancement oder Reflection aus
- **Referenzimplementierung** EclipseLink (früher: TopLink)
- Teil von Oracle Glassfish Server (**Glassfish** 4.0)
- EclipseLink benutzt nur Reflection

- ”The Jakarta Persistence API draws upon the best ideas from persistence technologies such as Hibernate, TopLink, and JDO.“

Grundbegriffe

- Grundbegriff: **Entity** – repräsentiert eine Tabelle in einer RDB
- Beschreibung von Entities durch **Annotationen**
- Eigene (portable) **SQL-Query** Sprache
- Unterstützt Java SE (**application managed persistence**) und EE (Enterprise Java Beans, **container managed persistence**)

- **Die wichtigsten JPA-Klassen:** package javax.persistence
- **EntityManager:** getTransaction, find, persist, refresh, remove, createQuery, clear ➔kümmert sich um persistente Objekte
- **EntityTransaction:** Transaktionen
- **Query:** SQL Abfragen erzeugen und ausführen



Packages #1

- Packages
  - javax.persistence
  - javax.persistence.spi
- Classes
  - Persistence
- Interfaces
  - EntityManagerFactory
  - EntityManager
  - EntityTransaction
  - Query
- Runtime Exceptions (~8)
  - RollbackException
- Annotations (~64)
  - Entity
  - Id
  - OneToOne
  - OneToMany
- Enumerations (~10)
  - InheritanceType
  - CascadeType
  - FetchType

Packages #2

- Das Package **javax.persistence** ist für die Entwicklerseite,
- das Pakage **javax.persistence.spi** definiert drei Interfaces für das Management der persistenten Klassen. *Der Entwickler hat mit dem Package javax.persistence.spi nichts zu tun.*
- Die Klasse **Persistence** ist quasi die Bootstrap-Klasse (im Container-Umfeld übernimmt dieser die Funktion von Persistence). Sie erzeugt eine EntityManagerFactory und in diesem Zug interne Instanzen von spi.PersistenceProvider, spi.PersistenceUnit und spi.ClassTransformer.
- Ausserdem wird ein neuer **ClassLoader** in der JVM installiert. Dieser ruft für die Entity-Klassen und alle Klassen, die mit Entities arbeiten, den **Transformer** auf, bevor diese Klassen in der JVM aktiv werden.
- Mit dem Transformer wird den geladenen Klassen zusätzliche Funktionalität für das Management der Persistenz hinzugefügt. Im Container-Umfeld übernimmt dieser die Aufgabe der Klasse Persistence.

Instanzierung des EnitiyManagers

- Ein Programm beginnt daher seine Persistenz-Aufgaben mit:

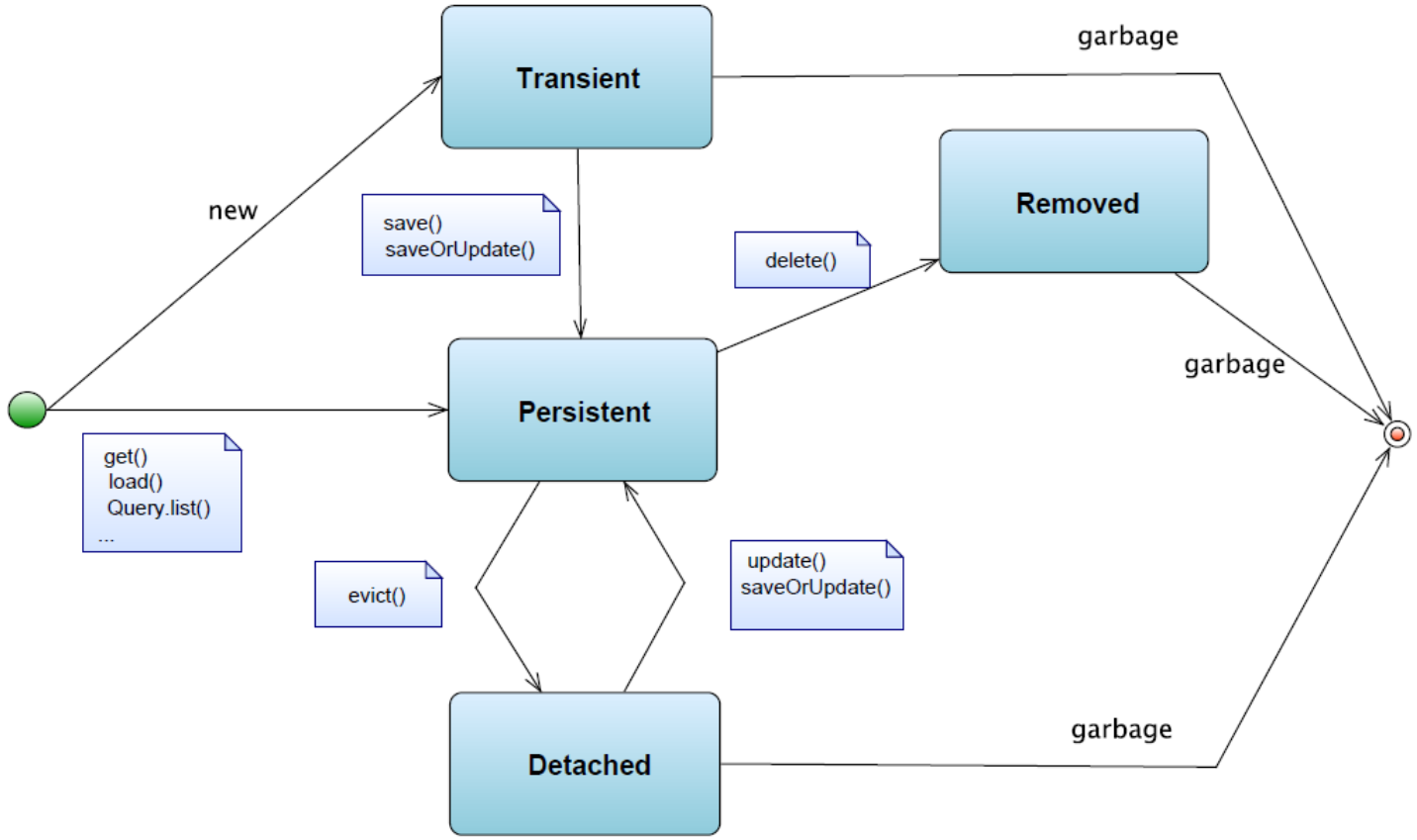
- `EntityManagerFactor emf = Persistence.createEntityManagerFactory(persistenceUnitName);`
- `EntityManager em = emf.createEntityManager()`

- im Container-Umfeld mit:
- `@PersistenceContext EntityManager em;`

Jakarta Persistence API

- **Mapping von Klassen**

Lebenszyklus von Objekten



O/R-Mapping - Objektidentität

- **Problem**
  - Effizienter Tabellenzugriff erfordert **eindeutigen Primärschlüssel**
  - Objekte können ***nicht*** **anhand ihres Wertes** unterschieden werden, sie besitzen Identität
  - **Datenbankensätze haben keine Identität**
- **Lösung**
  - ***Object Identifier*** (OID)
  - Jedem Datenbankensatz wird **eindeutige OID** zugeordnet
  - Objekte werden in der DB anhand ihres OIDs identifiziert

O/R-Mapping - Objektidentität

- **No Business Meaning:**
  - keine Attribute aus dem **modellierten Problembereich** als OIDs (d. h. als (Fremd-) Schlüssel)
  - bei **Attributen** oft **keine Eindeutigkeit** gegeben
  - im Prinzip kann sich jedes Attribut ändern, die Objektidentität bleibt aber unverändert
  - *Abhängig vom Typ eventuell ineffizient*

- **Wie eindeutig sollen die OIDs sein?**
  - für die **konkrete Klasse**?
  - innerhalb der **Klassenhierarchie**?
  - Über **alle Klassen** hinweg?

O/R-Mapping - Objektidentität

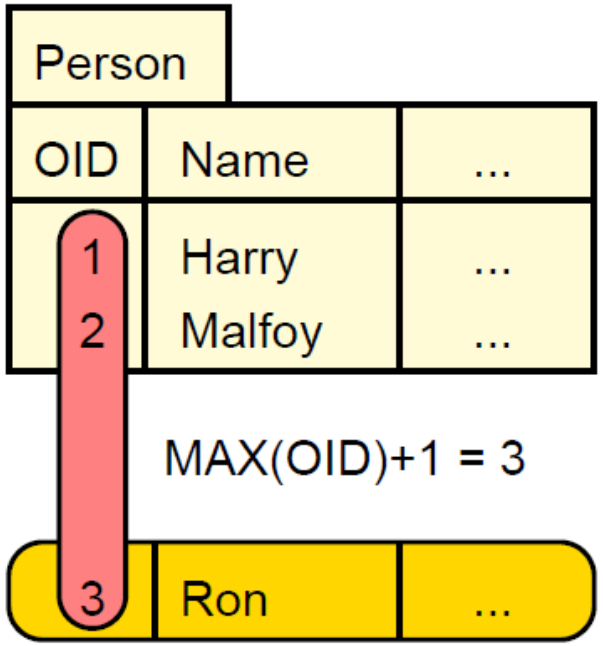
- Beispiel:

Leute : Person		
OID	Name	. . .
1	Harry	. . .
2	Malfoy	. . .

- neue Person Ron soll angelegt werden
- **Frage:** Welche OID soll (kann) vergeben werden?

Strategien zur OID-Zuordnung

MAX()+1 von OID-Spalte

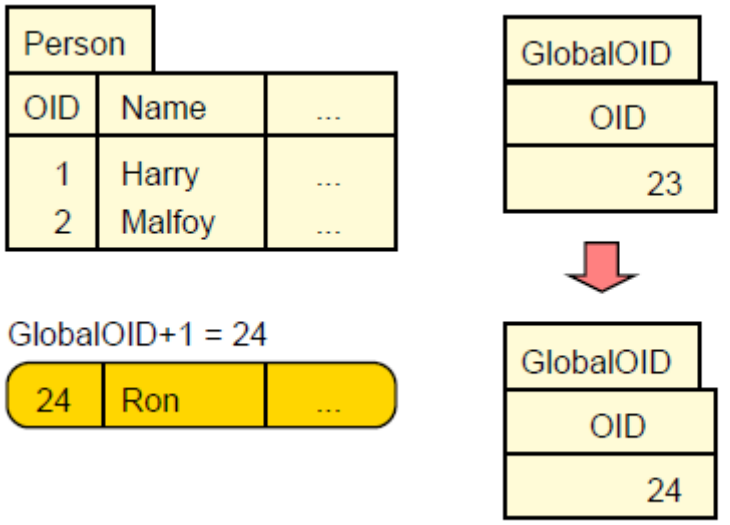


Bewertung:

- Lock auf Tabelle nötig
- nicht eindeutig über Tabellengrenzen

Strategien zur OID-Zuordnung

spezielle OID-Tabellen



Bewertung:

\+ systemweite Eindeutigkeit

\+ Datentabellen nicht gelocked

\- Lock auf OID-Tabelle ⇒ *Bottleneck*

Variation:

✓Eigene OID pro Tabellenname:

⇒ Tabelle mit | TableName | OID |

Strategien zur OID-Zuordnung

GUIDs/UUIDs (Globally/**U**niversally **U**nique **Id**s)

✓Berechne quasi eindeutige ID (128 Bit) aus

- MAC-Adresse, Datum, Zeit, oder
- (kryptographischem) Zufallszahlengenerator

Bewertung

\+ funktioniert

\+ Datentabellen nicht gelocked

\- proprietär

\- zu groß (kein Int/Long mehr)

Strategien zur OID-Zuordnung

High/Low Ansatz

- Zweiteilung der OID in HIGH- und
- LOW-Teil
  - ✓ HIGH zentral vergeben
  - ✓ LOW durch Anwendung selbst
  - ✓ Überlauf von LOW: neues HIGH

Bewertung

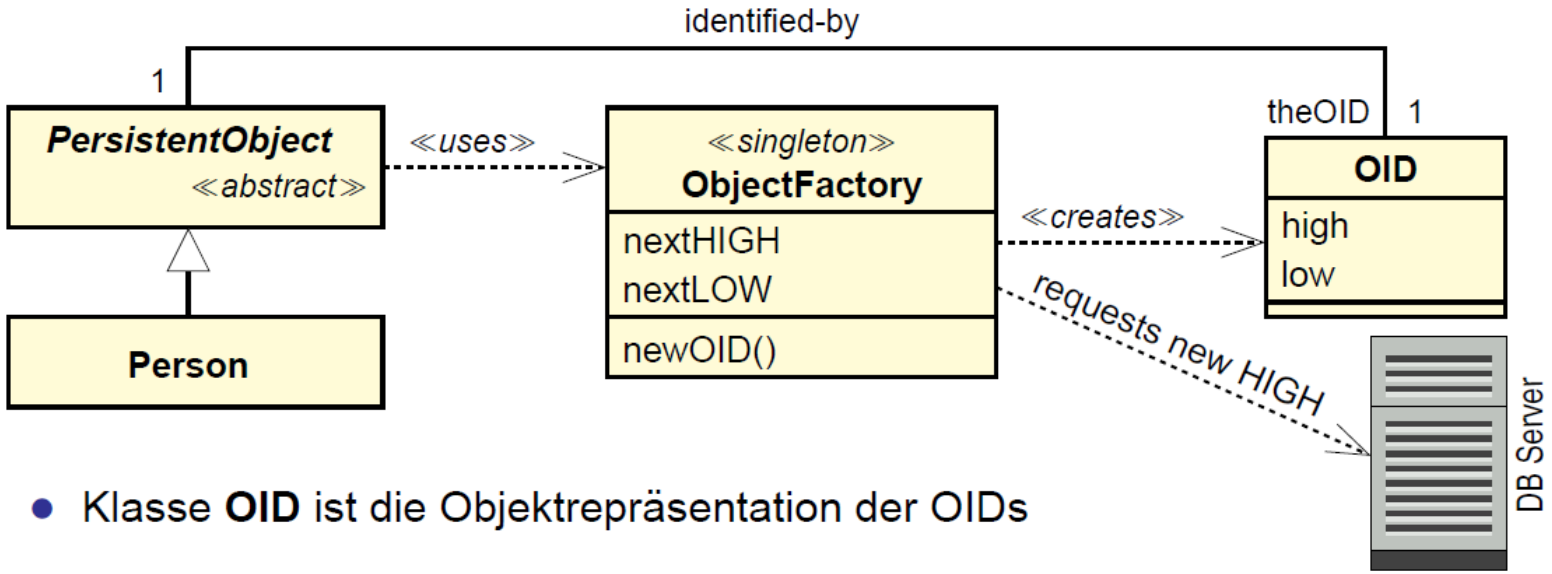
\+ zentrale OID-Vergabe nicht länger Bottleneck

\+ wenig OID-bezogener Netzwerk-traffik

\+ eindeutig für alle Klassen

Implementierung von High/Low

- Singleton **ObjectFactory** für Vergabe von OIDs zuständig
  - verwaltet nächste zu vergebende OID
  - besorgt bei Bedarf neues HIGH von einem Server



- Klasse **OID** ist die Objektrepräsentation der OIDs

Umsetzung des Datenmodells in Entities

EJB 3.2 - Java Persistence API (JPA 2.1)

Voraussetzungen:

- Markierung der Domänenobjekte
- Eindeutige Identifizierung jedes Objekts (Schlüssel)
- Definition der Objektbeziehungen
  - Definition von Regeln zum Speichern in/Laden aus einer Datenbank

Java Klasse – „noch keine Entity“

```
public class PollResult {
    private int ranking;

    private String name;

    public void setRanking(int ranking) {
        this.ranking = ranking;
    }
    public int getRanking() {
        return ranking;
    }
    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

Vergebene Punktzahl

Name (= eindeutiger Identifikator)

Klasse speichert das Ergebnis einer Abstimmung.

Entity Bean

```
@Entity
public class PollResult {

    private int ranking;

    @Id
    private String name;

    public void setRanking(int ranking) {
        this.ranking = ranking;
    }
    public int getRanking() {
        return ranking;
    }
    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

Markierung als Entity

Eindeutiger Identifikator (d.h. Schlüssel)

Eigenschaften einer Entity

- Muss einen **public-Konstruktor** ohne Parameter haben
  (Ist gar kein Konstruktor definiert, so gibt es automatisch einen Public-Konstruktor ohne Parameter).
- Kann von anderen Klassen **erben**
- Kann **abstrakt** sein
- Klasse darf nicht final, kein Interface und keine Enumeration sein und keine final-Methoden enthalten
- **Referenzierte Klassen** müssen auch **Entities** oder **serialisierbar** sein
- Annotation der Felder oder der Getter/Setter, Unterscheidung SE/EE
- Felder müssen private oder protected sein

Markierung für die Variablen

- **@Id:**
  Der Primärschlüssel für dieses Objekt, d.h. der Identifikator

- **@Transient:**
  Wird nicht mit abgespeichert, ist also beim Laden 0, null etc.

@id: Erlaubte Datentypen

- Alle **primitiven Typen**, String, int, long, etc. (aber nach Möglichkeit float und double vermeiden - wegen Genauigkeit der Kommastellen)
- Alle Wrapperklassen und serialisierbaren Typen (z.B. Integer, BigDecimal, Date, Calendar)
- **java.lang.Date** oder **java.sql.Date**
- …– byte[], Byte[], char[], Character[]
- Enumerations
- Beliebige weitere Entity-Klassen
- Collections von *Entities*, welche als Collection<>, List<>, Set<> oder Map<> deklariert sind.
- Viele Datenbanken unterstützen die „automatische Erstellung“ von Schlüsseln, z.B. über **Identity Columns**
- Dem Schlüsselfeld wird automatisch ein Wert zugewiesen, der nichts mit denDaten der Entity zu tun hat (d.h. z.B. „12567“ statt „Hans Wurst“)

@id: Nicht erlaubte Datentypen

- Alle Arten von Arrays außer die vorher genannten
- Collections von etwas anderem als Entities, also z.B. Wrapperklassen und andere serialiserbare Typen.

Enitity Identity - Primärschlüssel

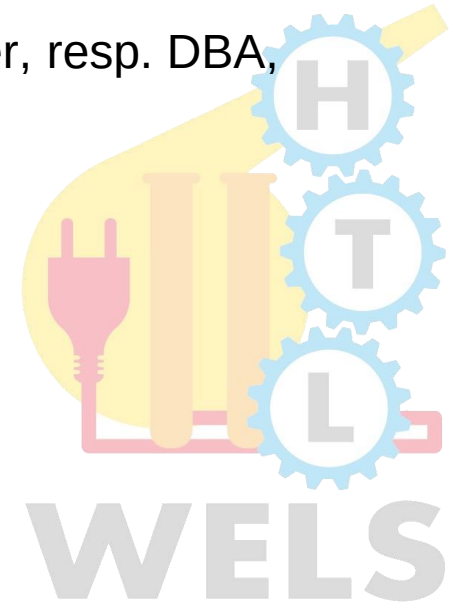
- Jede Entity-Klasse muss einen mit **@Id** bezeichneten Primärschlüssel besitzen.
- Primärschlüssel können in Zusammenarbeit mit der Datenbank generiert werden:

```
@Entity public class Message {
      @Id@GeneratedValue(strategy=IDENTITY )
public long id;
```

Strategien für die Identity

- **@GeneratedValue(strategy=STRAT)**

- Strategien sind Identity, Table, Sequence und Auto
  - **Identity und Sequence**: Das Datenbankssystem erzeugt eine fortlaufende Nummer. Sequence lehnt sich an die Oracle-Technologie an.
  - **Auto**: vom Framework gewählte Strategie. Meist gleichbedeutend mit Identity.
  - **Table**: Erzeugung via Tabelle in Datenbank. Diese Tabelle wird durch den Entwickler, resp. DBA, erstellt und initialisiert.



Erzeugung einer Tabelle

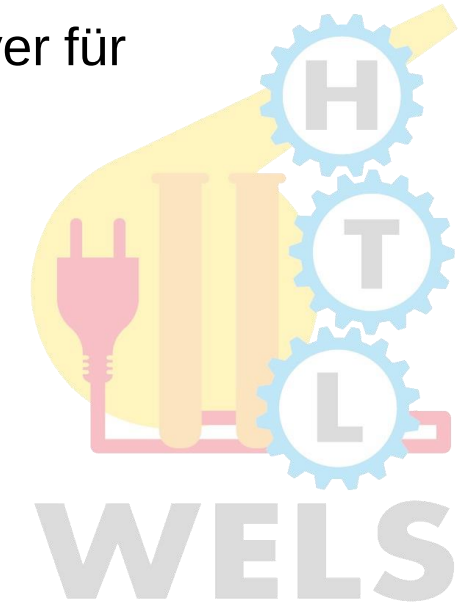
```
public class Message {
    @TableGenerator(
        name="MyIdGenerator",
        table="KeyStore",
        pkColumnName="keyName",
        valueColumnName="keyValue",
        pkColumnValue="MessageId",
        initialValue=1000,
        allocationSize=1)
@Id @GeneratedValue( strategy=GenerationType.TABLE,
generator="MyIdGenerator")
protected long id;
// ...
}
create table KeyStore (
    keyName nvarchar( 32 ) not null unique,
    keyValue bigint not null default 1
);
insert into KeyStore ( keyName, keyValue )
values ( 'MessageId', 1000 );
```

Java Type / SQL-Type Mapping #1

- Implizit durch JDBC 3.0, Data Type Conversion Table, definiert.
- Explizit durch die @Column Annotation, z.B.

```
@Column( name="sender", columnDefinition="VARCHAR(255) NOT NULL" )
protected String sender;
```

- Produktspezifisch durch Framework (Hibernate) oder im JDBC-Driver für die jeweilige Datenbank.



Java Type / SQL-Type Mapping #2

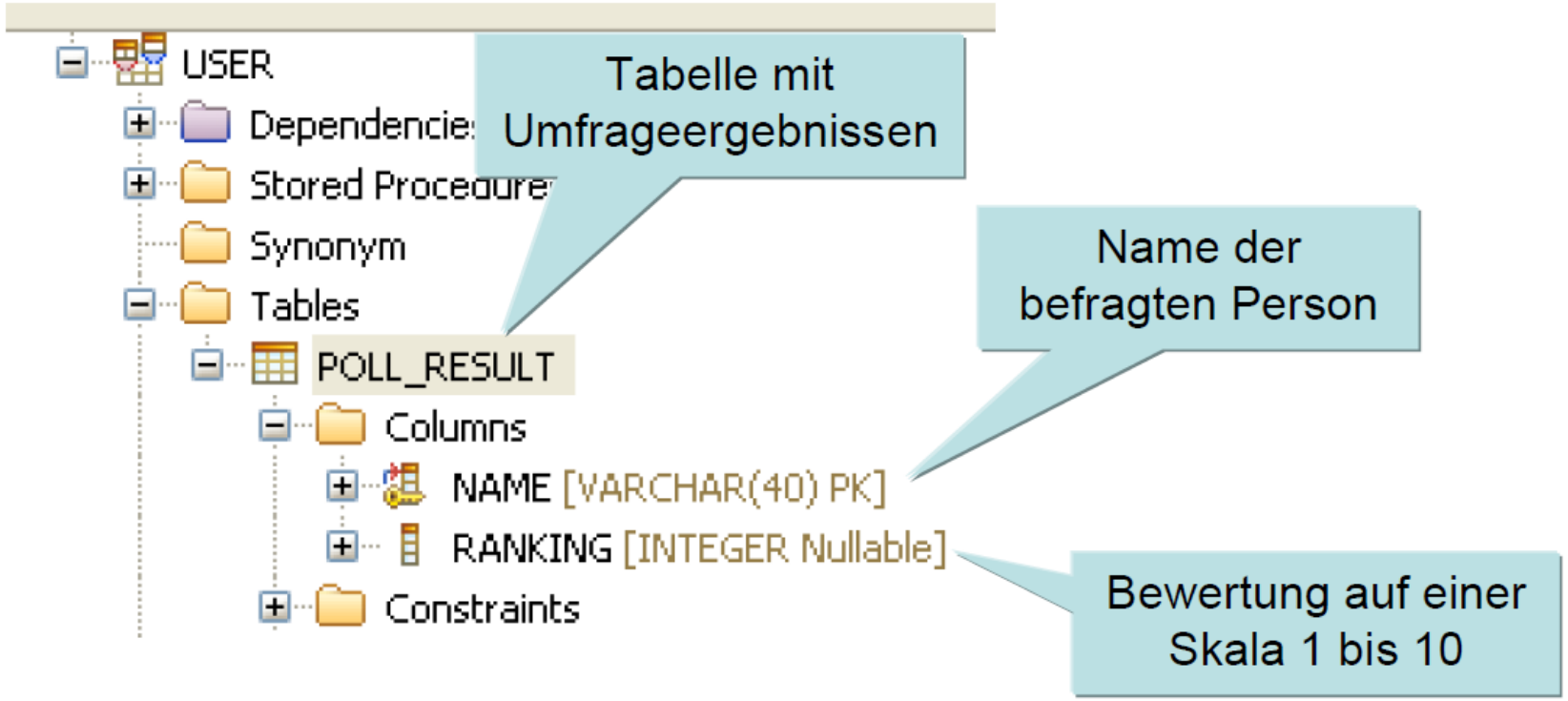
JDBC Type	Java Type
CHAR	String
VARCHAR	String
LONGVARCHAR	String
NUMERIC	java.math.BigDecimal
DECIMAL	java.math.BigDecimal
BIT	boolean
BOOLEAN	boolean
TINYINT	byte
SMALLINT	short
INTEGER	int
BIGINT	long
REAL	float
FLOAT	double
DOUBLE	double
BINARY	byte[]
VARBINARY	byte[]
LONGVARBINARY	byte[]
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp

Erstellen einer Identity Column

```
CREATE TABLE person2 (
        ID INTEGER NOT NULL primary key GENERATED ALWAYS AS IDENTITY
             (START WITH 1, INCREMENT BY 1),
        name VARCHAR(40) not null,
        email VARCHAR(40),
        telephone VARCHAR(40),
        fax VARCHAR(40)
        );
```

Automatischer Tabellenwert, beginnt bei 1, wird jedesmal um 1 erhöht

Datenbank Umfrageergebnisse



Entity Klasse

```
@Entity
@Table(name="person2")
public class IDPerson {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    @Column(name="id")
    private
    int id;

    @Column(name="name")
    private String name;

    @Column(name="email")
    String email;

    @Column(name="telephone")
    private
    String telephone;

    @Column(name="fax")
    private
    String fax;
```

Einziger Unterschied zu normaler Entity.

Aber: Die Spalte muss in der DB entsprechend eingerichtet sein!

Datenbanksicht

IDPerson.java | GlassFish v3 - Error | GlassFish v3 - Error | http://localhost:808 | PERSON2

ID [INTEGER]	NAME [VARCHAR(40)]	EMAIL [VARCHAR(40)]	TELEPHONE [VARCHAR(40)]	FAX [VARCHAR(40)]
1	Carlos		NA	01234/56789
<new row>				

Mehrmals identischer Aufruf

sqlscrap.sql | IDPerson.java | IIDPersonManipulator | PERSON2 | http://localhost:808

ID [INTEGER]	NAME [VARCHAR(40)]	EMAIL [VARCHAR(40)]	TELEPHONE [VARCHAR(40)]	FAX [VARCHAR(40)]
1	Carlos		NA	01234/56789
2	Carlos		NA	01234/56789
3	Carlos		NA	01234/56789
4	Carlos		NA	01234/56789
<new row>				

Persistence Unit #1

- Eine **Persistence Unit** ist eine *logische* Einheit von Entities.

- Sie wird beschrieben durch:
  - Einen **Namen**
  - Die zu dieser Unit gehörenden **Entity-Klassen**
  - Angaben zum **Persistence Provider**
  - Angabe zum **Transaktionstyp**
  - Angaben zur **Datenquelle**
  - Weitere **Properties**

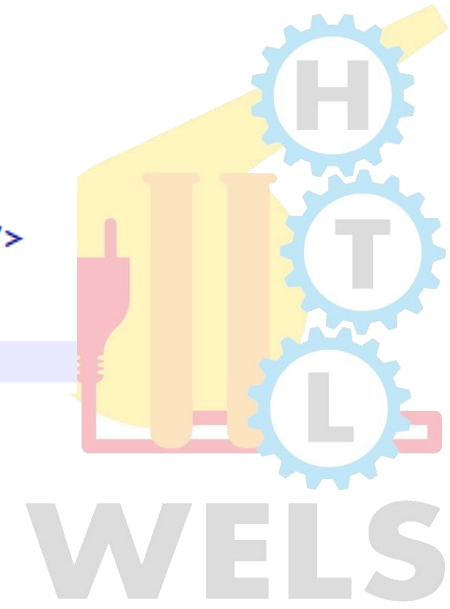
- Technisch wird die Beschreibung einer Persistence Unit in der Datei META-INF/persistence.xml abgelegt.

Persistence Unit #2

- Der **Name** ist eine Identifikation, um in der Applikation dafür eine EntityManagerFactory erstellen zu können.
- Die Entity-Klassen werden als **Information** für den **Class Transformer** und die Generierung der SQL-Befehle angegeben.
- Der **PersistenceProvider** ist die implementationsspezifische "**Urklasse**" für das **Persistenzmanagement** in dieser Persistence Unit. Sie liefert die **EntityManagerFactory** zurück.
- Der **Transaktionstyp** (transaction-type) wird als JTA oder RESOURCE_LOCAL angegeben. Eine JTA Persistence Unit muss u.A. verteilte Transaktionen über eine XA-Schnittstelle erlauben.
- Die Datenquelle kann als JNDI-Name angegeben werden im Rahmen einer J2EE-Umgebung. Je nachdem ob die Datenquelle JTA-fähig ist (verteiltes Transaktionsmanagement) verwendet man den Tag <jta-data-source> oder <non-jata-data-source>. Falls nicht mit JNDI gearbeitet wird, können in den Properties die üblichen JDBC-Verbindungsparameter angegeben werden.

persistence.xml

```
<persistence>
        <persistence-unit name="PMQ" transaction-type="RESOURCE_LOCAL">
            <provider>org.hibernate.ejb.HibernatePersistence</provider>
            <class>pmq.PMQ</class>
            <class>pmq.Message</class>
            <non-jta-data-source>jdbc/MyPMQDB</non-jta-data-source>
            <jta-data-source>jdbc/MyPMQDB</jta-data-source>
            <properties>
                <property name="hibernate.connection.driver_class"
                value="com.microsoft.sqlserver.jdbc.SQLServerDriver"/>
                <property name="hibernate.connection.username" value="hibuser"/>
                <property name="hibernate.connection.password" value="hibuser"/>
                <property name="hibernate.connection.url"
                value="jdbc:sqlserver://localhost;databaseName=hib_db"/>
                <property name="dialect" value="org.hibernate.dialect.SQLServerDialect"/>
            </properties>
        </persistence-unit>
</persistence>
```



Persistence Context

- Der Persistence Context definiert das **physische Umfeld** von Entities zur **Laufzeit**:
  - Die **Menge aller Managed Entities** in der Applikation
  - Der **Entity Manager** für diese Entities
  - Die laufende **Transaktion**
  - Der **Contexttyp**

```
@PersistenceContext(type=PersistenceContextType.EXTENDED)
public class MessageProducer {
// ...
EntityManagerFactory emf =
    Persistence.createEntityManagerFactory("PMQ");
    EntityManager em = emf.createEntityManager();
    Transaction tx = em.getTransaction();
// ...
}
```

Persistence Context - Contexttyp

- Contexttyp **TRANSACTION**
  - Lesender und schreibender Zugriff nur innerhalb der Transaktion.
  - Gelesene Objekte sind nach der Transaktion im Zustand detached.
  - Wiedereinkopplung in eine Transaktion mit merge().

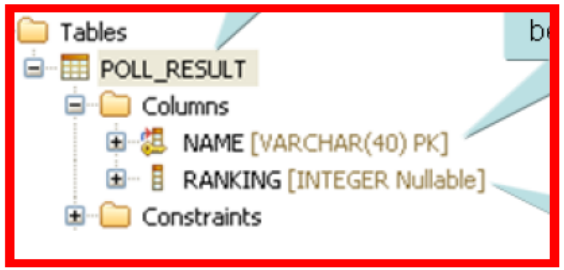
- Contexttyp **EXTENDED**
  - Alle Objekte sind lesend und schreibend zugreifbar.
  - Modifikationen finden lokal statt.
  - Effekt von persist(), remove() usw. wird aufbewahrt.
  - Propagation von Efffekten und Änderungen in die DB aber nur, wenn **begin**()/**commit**() ausgeführt wird.

Erzeugen persistenter Objekte (Beispiel)

```
// ...
EntityManagerFactory emf =
Persistence.createEntityManagerFactory("PMQ");
EntityManager em = emf.createEntityManager();
EntityTransaction tx = em.getTransaction();

tx.begin();
Message m = new Message( ... );
em.persist( m );
tx.commit();
// ...
```

Abbildung auf Datenbank

```java
@Entity
@Table(name="POLL_RESULT")
public class PollResult {
    @Column(name="RANKING")
    private int ranking;

    @Id
    @Column(name="NAME", nullable=false, length=40)
    private String name;
```



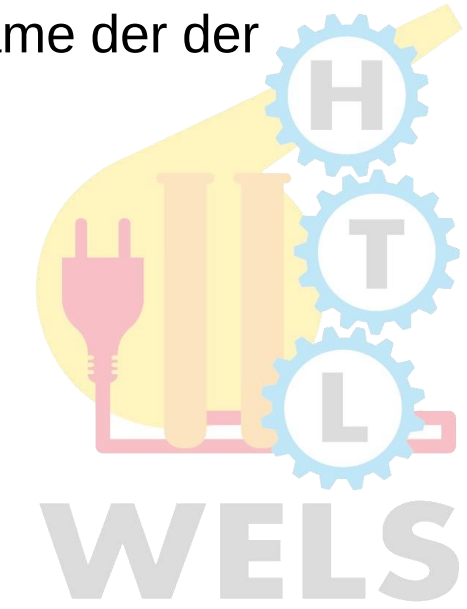
Annotationen für Abbildung auf Tabellen #1

- **@Table**

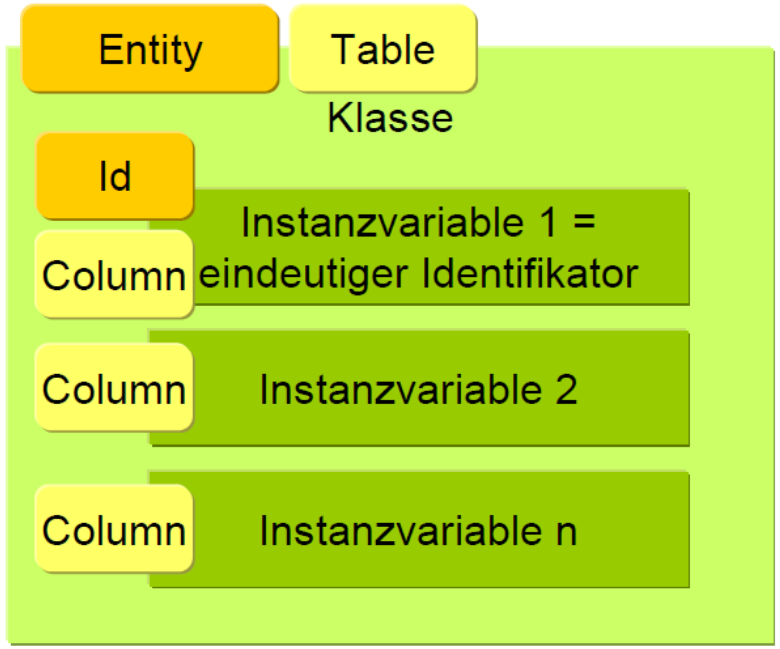
- Kennzeichnet die zu verwendende Tabelle
- Verwendet man diese Annotation nicht, so wird die Tabelle mit dem gleichen Namen wie die Entity verwendet
- *Optionale Parameter: catalogue, schema, uniqueConstraints*

Annotationen für Abbildung auf Tabellen #2

- **@ Column**
- name: Name der Spalte in der angegebenen Tabelle
- *table: Nur relevant wenn man die Annotation @SecondaryTable verwendet hat*
- nullable: Darf in der Spalte ein Nullwert stehen?
- unique: Muss der Wert in der Tabelle einmalig sein?
- length, precision, scale: Nur bei bestimmten Datentypen relevant Fehlt der Name der der Column wird der Feldname als Spaltenname genommen.



Zusammenfassung Enitites



Entity Annotationen *ORM Annotationen*

Entity Manager erzeugen

```
@PersistenceContext(unitName="test")
private EntityManager entityManager;
```

Wird durch Dependency Injection in die SessionBean-Klasse eingefügt

persistence.xml (liegt im Verzeichnis META.INF)

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence>
  <persistence-unit name="test">
    <jta-data-source>jdbc/sample</jta-data-source>
    <properties>
    </properties>
  </persistence-unit>
</persistence>
```

unitName

Name der Datenbank (s. Anhang)

Objekte schreiben

```
public void createPollEntry(String name, int value) {
    PollResult result = new PollResult();
    result.setName(name);
    result.setRanking(value);
    entityManager.persist(result);
}
```

Objekterzeugung

Parameter ist ein Objekt einer als Entity markierten Klasse

Objekt lesen

```
public PollResult recoverPollEntry(String name) {
    return (PollResult) entityManager.find(PollResult.class, name);
}
```

Daten aus der Tabelle auslesen über Primärschlüssel

Gesamtes Beispiel

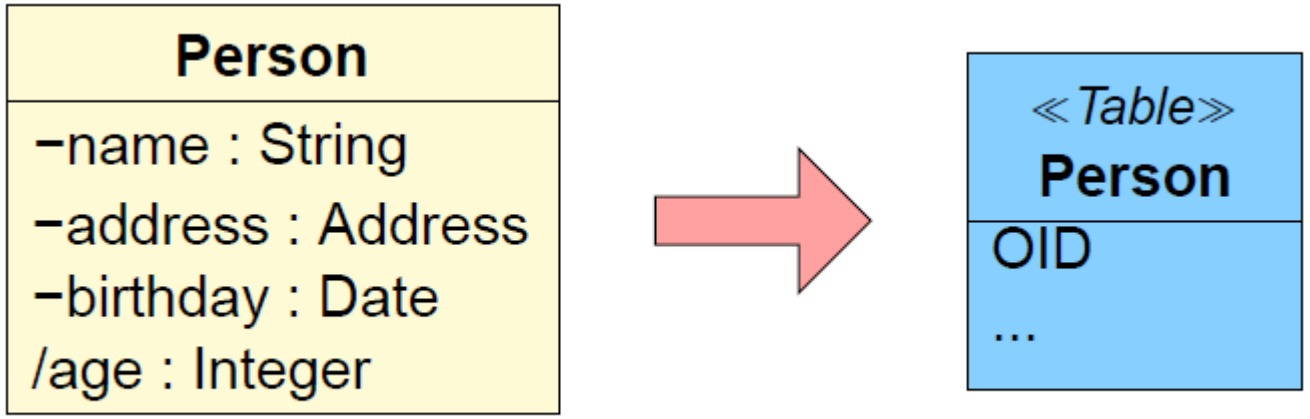
```
@Stateless
public class PollEntryCreator implements IPollEntryCreator {

    @PersistenceContext(unitName="test")
    private EntityManager entityManager;

    public PollEntryCreator() {
    }

    public void createPollEntry(String name, int value) {
        PollResult result = new PollResult();
        result.setName(name);
        result.setRanking(value);
        entityManager.persist(result);
    }

    public PollResult recoverPollEntry(String name) {
        return (PollResult) entityManager.find(PollResult.class, name);
    }

    public AggregatedPollResults showAggregatedResults() {
        AggregatedPollResults results = new AggregatedPollResults();
        return results;
    }

}
```

Abbildung von Klassen

- **Prinzip**

  ➔Klassen werden auf Tabellen abgebildet

- die erste(n) Spalte(n) sind für die OID
- dann folgen die Attribute



Lazy Loading

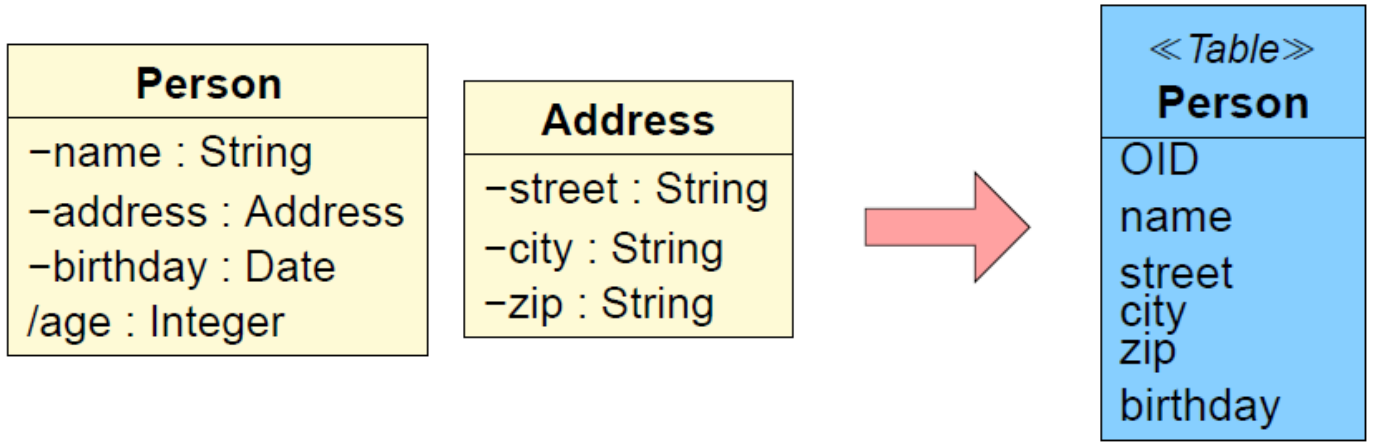
- **Problem**
- Objekte sind mit (teilweise vielen) anderen Objekten assoziiert
  - Sollen diese alle **gleichzeitig** mit materialisiert werden?
  - Werden von der Anwendung vielleicht **gar nicht benötigt**!
  - Haben womöglich gar nicht im Hauptspeicher Platz!
- **Lösung**
  - Objekte erst materialisieren, wenn auf sie zugegriffen wird:
    ➔ Proxy Pattern

Abbildung von Attributen

- **Prinzip**

  ➔Attribute werden in (≥0) Spalten abgebildet

- abgeleitete Attribute: werden nicht in die DB abgebildet
- einfache Attribute: eine Spalte
- objektwertige Attribute *ohne eigene Identität*: eine Spalte pro Attribut (rekursive Definition)



Objekt ID

- Ein neues Objekt bekommt erst eine ID, wenn es das erste Mal **physisch** in die Datenbank transportiert wird.

```
Message m = new Message(...)

pmq.append( m );
System.out.println( m.getId() )

em.flush()
System.out.println( m.getId() )
```

- Beim ersten System.out.println ist die OID == 0;

Jakarta Persistence API

- **Mapping von Klassenhierarchien**

Abbildung von Klassenhierarchien - Beispiel

- Wie kann man nachstehende Klassenhierarchie auf DB-Tabellen abgebildet werden?

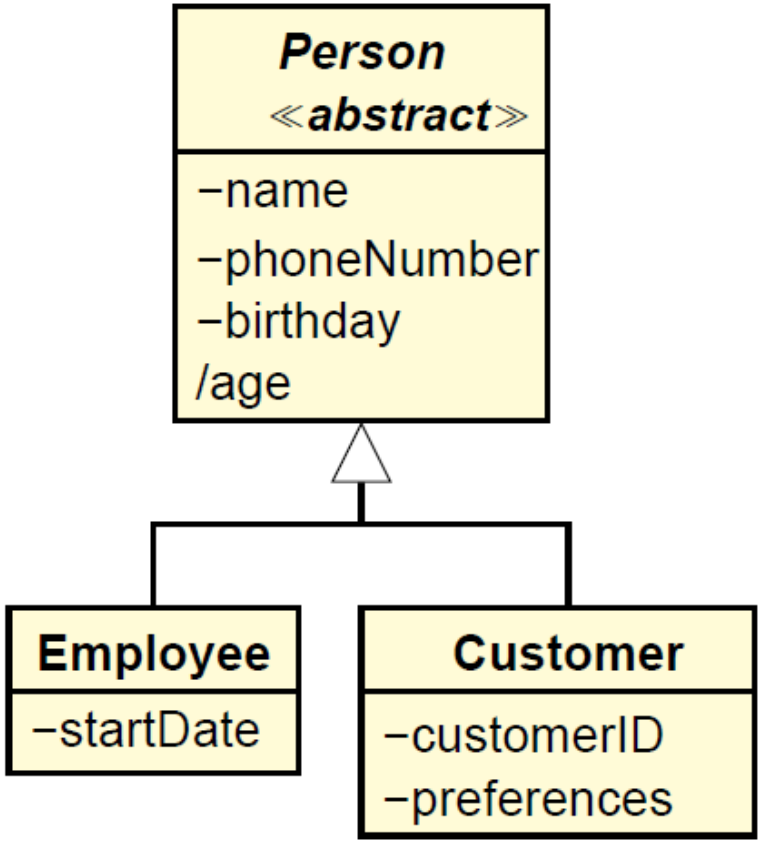


Abbildung von Klassenhierarchien

3 Möglichkeiten:

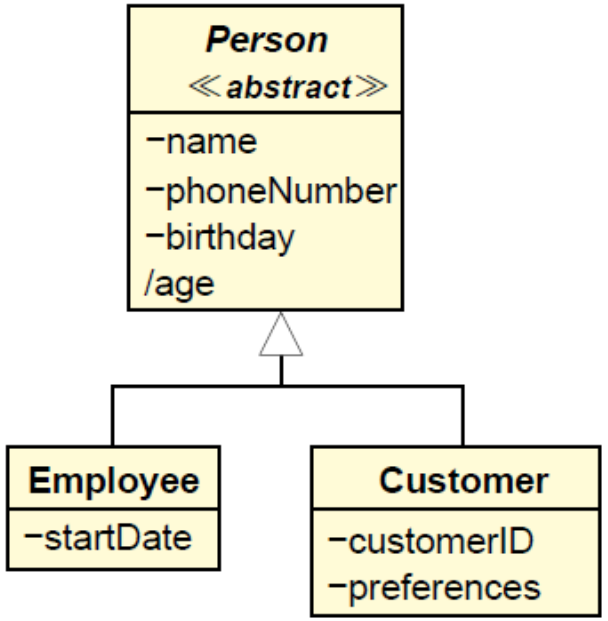
1. eine einzige Tabelle für die gesamte Klassenhierarchie **(SINGLE TABLE)**
2. eine (vollständige) Tabelle pro konkreter Klasse **(TABLE PER CLASS)**
3. eine (partielle) Tabelle pro Klasse **(JOINED)**

Abbildung von Klassenhierarchien

Möglichkeit 1: Eine Tabelle für die *gesamte Hierarchie* **(SINGLE TABLE).**

Bewertung

+ einfach
+ unterstützt Polymorphismus
- erhöhte Kopplung in der DB
- Platzverschwendung
- Null muss zugelassen sein



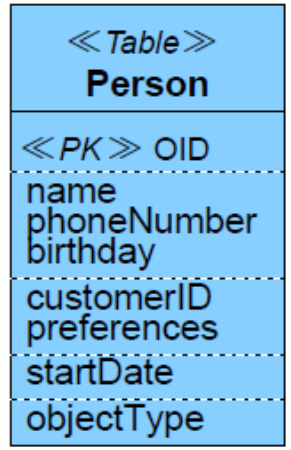
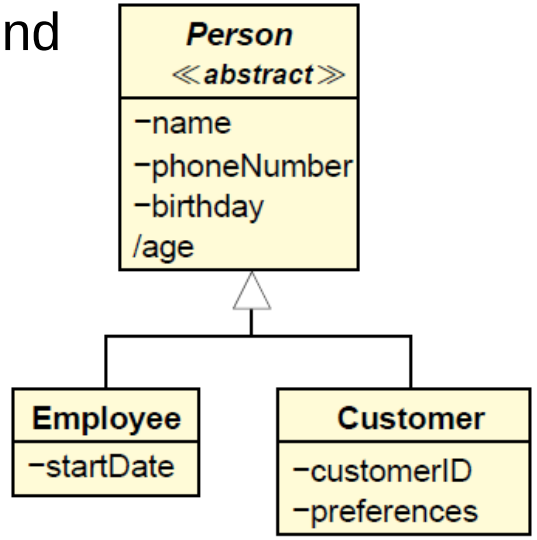


Abbildung von Klassenhierarchien

Möglichkeit 2: Eine Tabelle für jede *konkrete Klasse* **(TABLE PER CLASS).**

Bewertung

+ einfacher Zugriff auf die Attribute einer Klasse
- Änderungen an oberster Klasse ➔ viele Tabellen ändern
+ Platzsparend



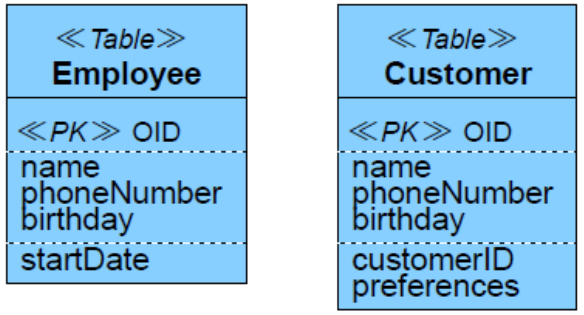
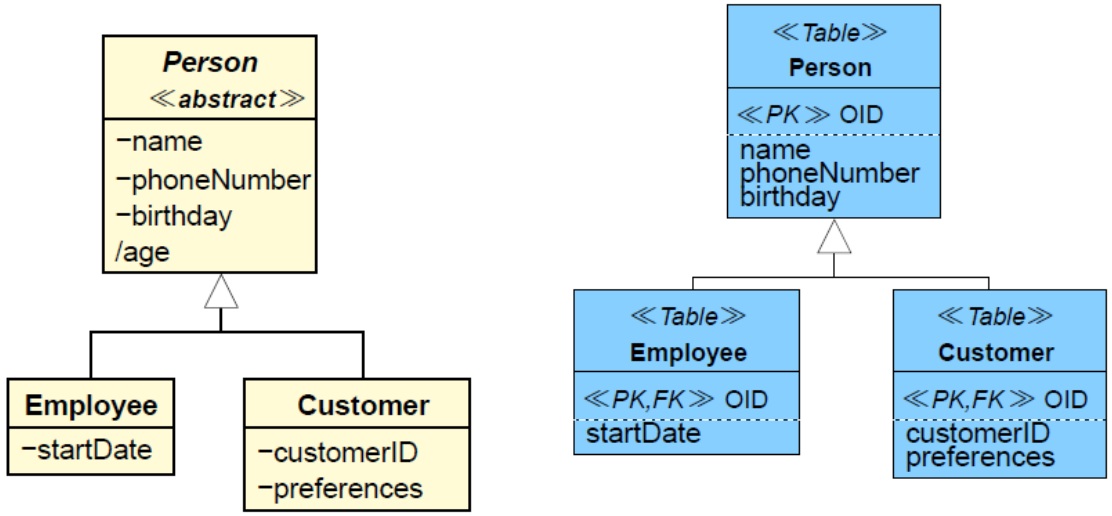


Abbildung von Klassenhierarchien

Möglichkeit 3: Eine partielle Tabelle für jede *Klasse* **(JOINED).**

Bewertung:

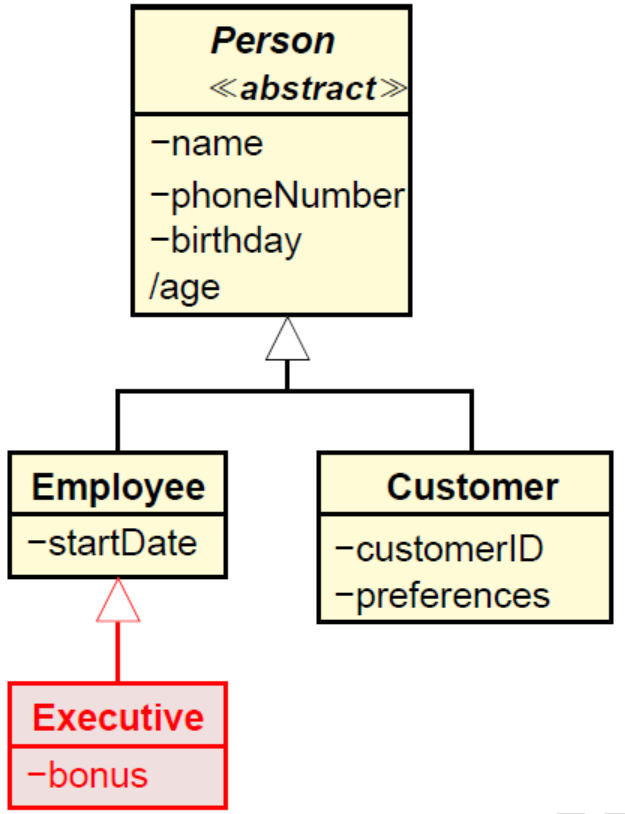
+ entspricht OO-Modell, sehr gute Unterstützung für Polymorphismus
+ leichtes Ändern/Hinzufügen von Klassen, Platzsparend
- viele Tabellen pro Klasse ⇒ längere Zugriffszeiten, Konsistenzproblem



Erweiterung der Klassenhierarchie

Die Klassenhierarchie wird um die Klasse Executive erweitert

Wie viel Änderungsaufwand fällt bei den einzelnen Modellierungen an?

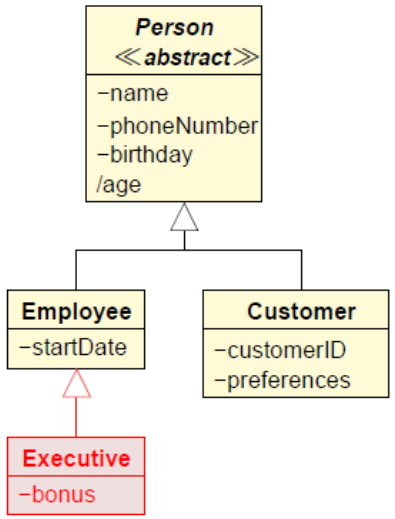


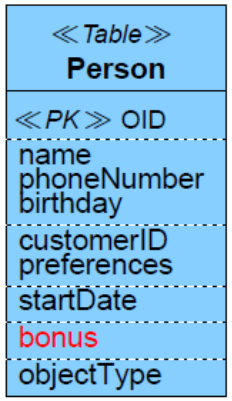
Erweiterung der Klassenhierarchie

Möglichkeit 1: Eine Tabelle für die *gesamte Hierarchie* **(SINGLE TABLE).**

Bewertung:

+ nur eine neue Spalte: sehr wenig Änderungsaufwand
- Aber: Tabelle muss geändert werden!
- Platzverschwendung



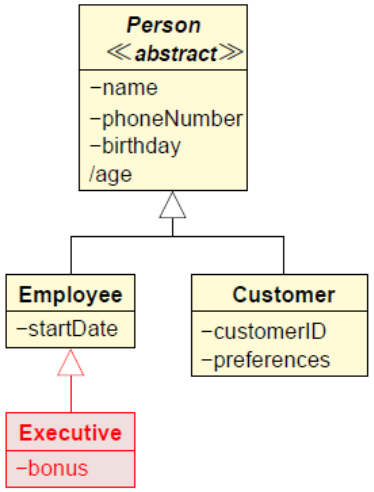


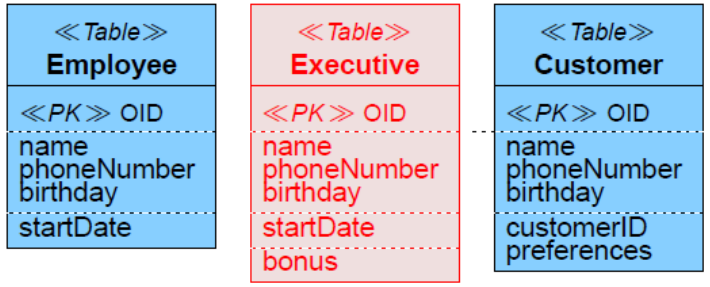
Erweiterung der Klassenhierarchie

Möglichkeit 2: Eine Tabelle für jede *konkrete Klasse*
(TABLE PER CLASS).

Bewertung:

+ nur eine neue Tabelle: wenig Änderungsaufwand, platzsparend
- komplizierte Rollenverwaltung: Kann ein Customer auch gleichzeitig ein Executive sein?



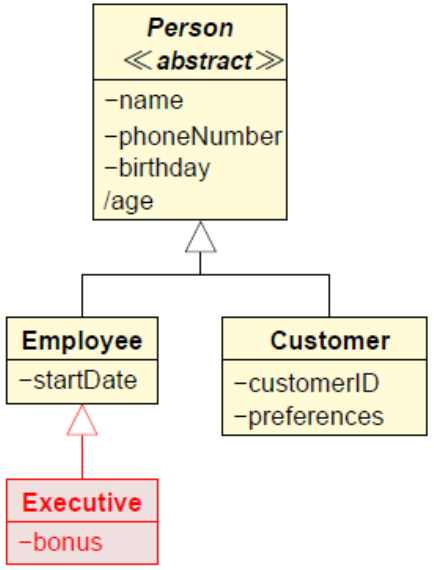


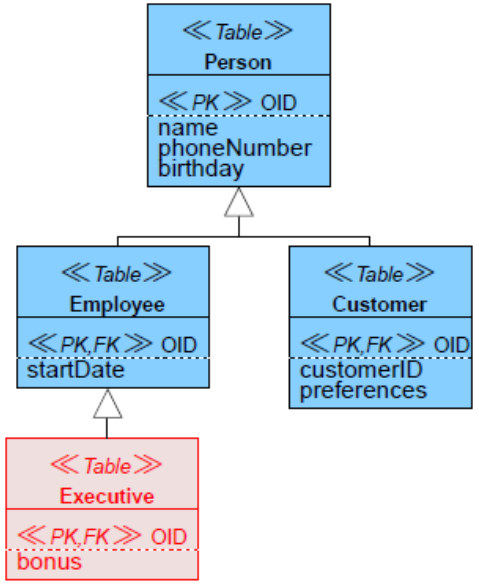
Erweiterung der Klassenhierarchie

Möglichkeit 3: Eine partielle Tabelle für jede *Klasse* **(JOINED).**

Bewertung:

+ einfache Änderung, sehr platzsparend
- Zugriffe auf Executive komplex und langsam





Vergleich der drei Ansätze

Kriterien	Tabelle für gesamte Hierarchie	Tabelle pro konkreter Klasse	Tabelle pro Klasse
Implementierung	Einfach	Mittel	Schwierig
Datenzugriff	Leicht	Leicht	Mittel–Schwer
Kopplung	Sehr hoch	Hoch	Niedrig
Geschwindigkeit	Schnell	Schnell	Mittel
Polymorphismus	Mittel	Niedrig	Hoch

Vererbungshierarchien - Grundsätzliches

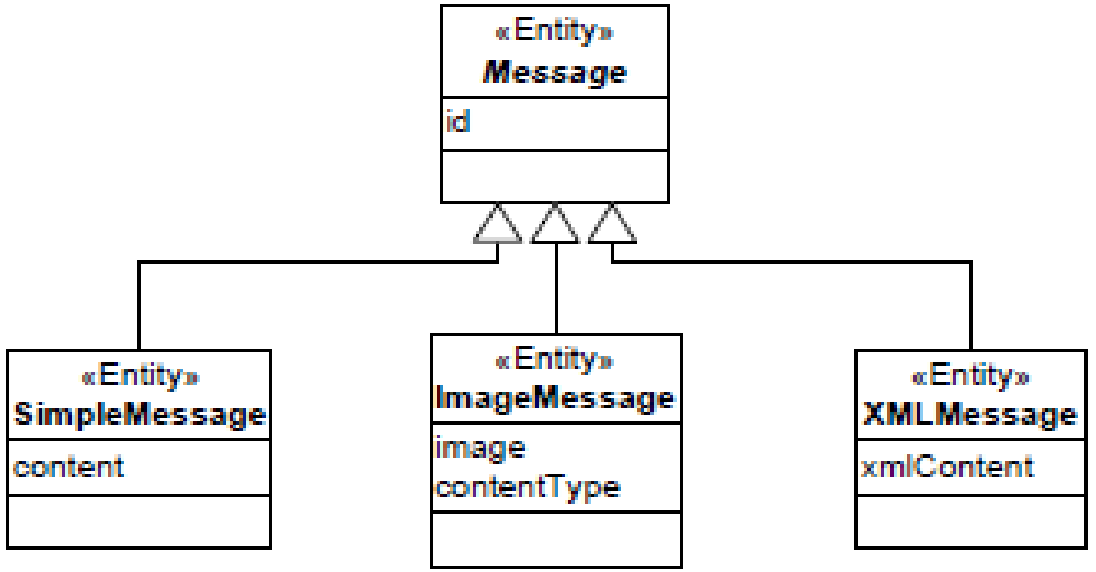
- Vererbungshierarchien können problemlos verwendet und abgebildet werden.
- **Klassen** können **abstrakt** oder **konkret** sein.
- Alle Klassen in der Vererbungshierachie müssen den **Primärschlüssel der Basisklasse** verwenden (erben).
- Es gibt **vier Mappingstrategien** auf die Datenbank:
  - Eine *einzige* Tabelle für die gesamte Verbungshierarchie
  - Eine Tabelle für jede *konkrete* Klasse
  - Eine Tabelle für *jede* Klasse
  - Mapped Superclass

Vererbung - Strategien

Die Strategie wird in der Basisklasse angegeben mit:

- **@Entitiy @Inheritance(strategy=SINGLE_TABLE)**
- **@Entitiy @Inheritance(strategy=TABLE_PER_CLASS)**
- **@Entitiy @Inheritance(strategy=JOINED)**
- **@MappedSuperclass**

Vererbung - Beispiel



Vererbung, SINGLE_TABLE Mapping

```java
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn( name="messageType" )
public abstract class Message {
@Id @GeneratedValue protected long id;
// ...
@Entity
@DiscriminatorValue("SM")
public class SimpleMessage extends Message {}
@Entity

@DiscriminatorValue("IM")
public class ImageMessage extends Message {}
```

```sql
create table "Message"
(
    id bigint not null default primary key,
    messageType varchar( 64 ) not null,
    sender nvarchar( 64 ) not null,
    receiver nvarchar( 64 ) not null,
    priority int not null,
    date datetime not null,
    content nvarchar( max ) null,
    image varbinary( max ) null,
    contentType nvarchar( 64 ) null,
    description nvarchar( 255 ) null,
    xmlContent xml null
);
```

Vererbung, JOINED Mapping

```java
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
public abstract class Message {
@Id @GeneratedValue protected long id;
// ...

@Entity
public class SimpleMessage extends Message {}

@Entity
public class ImageMessage extends Message {}
```

```sql
create table "Message" (
    id bigint not null identity primary key,
    sender nvarchar( 64 ) not null,
    receiver nvarchar( 64 ) not null,
    priority int not null,
    date datetime not null,
);
create table "SimpleMessage" (
    id bigint not null primary key,
    content nvarchar( max ) not null,
    foreign key ( id ) references Message( id )
);
create table "ImageMessage" (
    id bigint not null primary key,
    image varbinary( max ) not null,
    contentType nvarchar( 64 ) not null,
    foreign key ( id ) references Message( id )
);
create table "XMLMessage" (
    id bigint not null primary key,
    xmlContent xml not null
    foreign key ( id ) references Message( id )
);
```

Jakarta Persistence API

- **Abbildung von Objektbeziehungen**

Abbildung von Objektbeziehungen

- Standardbeziehungen
  - 1:1-Beziehungen (unidirektional, bidirektional)
  - 1:n und n:1 Beziehungen (unidirektional, bidirektional)
  - n:m Beziehungen (unidirektional, bidirektional)

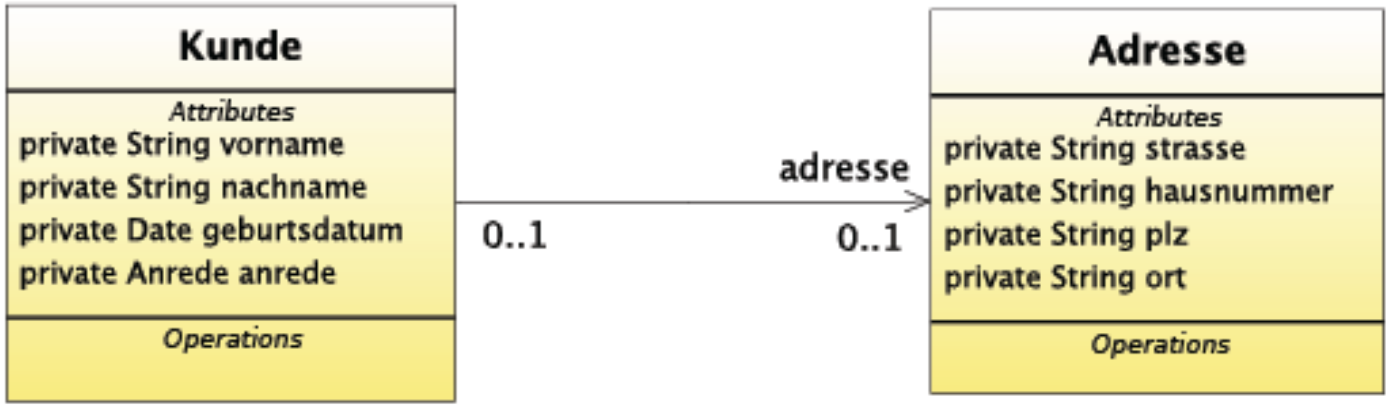
- Andere Beziehungsklassen (exemplarisch)
  - Rekursive Beziehungen

Abbildung von Objektbeziehungen

1:1- BEZIEHUNGEN

1:1 Beziehung (unidirektional)

- Beispiel



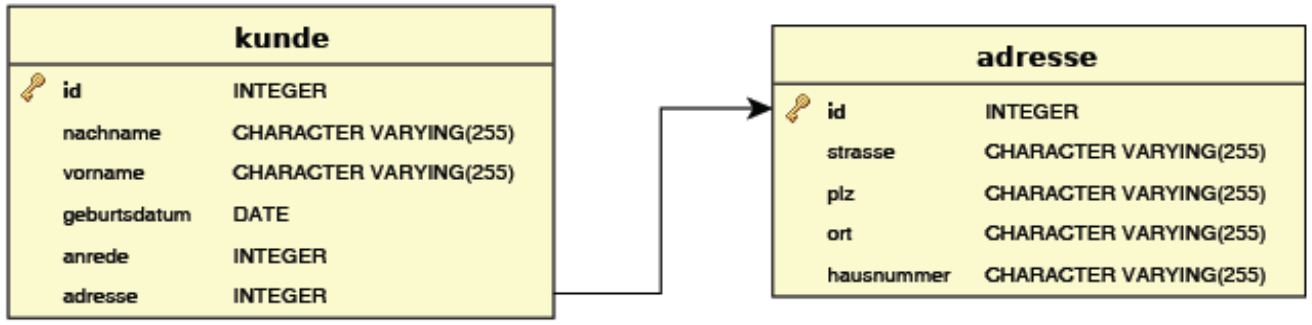
1:1 Beziehung (unidirektional)

- Mapping

```
@Entity
public class Kunde {

  @Id @GeneratedValue
  private Integer id;
  private String vorname;
  private String nachname;
  @Temporal(TemporalType.DATE)
  private Date geburtsdatum;
  private Anrede anrede;
  @OneToOne
  @JoinColumn(name = "adresse")
  private Adresse adresse;
```

```
@Entity
public class Adresse {

  @Id @GeneratedValue
  private Integer id;
  private String strasse;
  private String hausnummer;
  private String plz;
  private String ort;
  ...
```

- Datenbank



1:1 Beziehung (unidirektional)

- Objekt erzeugen und persistieren

```
Kunde kunde = new Kunde("Max", "Mustermann", ...);
Adresse adresse = new Adresse("Goethestrasse", ...);
kunde.setAdresse(adresse);
em.persist(kunde);
em.persist(adresse);
```

- Objekt suchen

```
Kunde kunde = em.find(Kunde.class, 1);
System.out.println("Kunde " + kunde +
                   " wohnt in " + kunde.getAdresse());
```

1:1 Beziehung (unidirektional)

- Jeder Kunde muss eine Adresse haben

```
@OneToOne(optional = false)
@JoinColumn(name = "adresse")
private Adresse adresse;
```

  - Besonderheit Eclipselink (Existenz der Beziehung wird nicht geprüft), daher

```
@OneToOne(optional = false)
@JoinColumn(name = "adresse", nullable = false)
private Adresse adresse;
```

1:1 Beziehung (unidirektional)

- @OneToOne Attribute

@OneToOne(...)			
Attribut	**Typ**	**Default**	**Beschreibung**
cascade	Cascade-Type[]	—	Zu kaskadierende Operationen auf der Zielseite der Assoziation
fetch	FetchType	EAGER	Ladezeitpunkt der Assoziation (EAGER, LAZY)
mappedBy	String	—	Name des Property der Beziehung auf der Eigentümerseite
optional	boolean	true	Angabe, ob Assoziation optional ist
orphan-Removal	boolean	false	Angabe, ob Orphan gelöscht werden soll
target-Entity	Class	void.class	Klasse des Ziels der Assoziation

1:1 Beziehung (unidirektional)

- @JoinColumn Attribute

@JoinColumn(...)			
Attribut	**Typ**	**Default**	**Beschreibung**
columnDefinition	String	—	Zusätzliches SQL-Fragment zur Definition der Spalte
insertable	boolean	true	Angabe, ob die Spalte in Insert-Anweisungen verwendet werden soll
name	String	—	Der Name der Fremdschlüsselspalte
nullable	boolean	true	Angabe, ob die Fremdschlüsselspalte Nullwerte zulässt
referencedColumnName	String	—	Name der Spalte, die durch diese Fremdschlüsselspalte referenziert wird
table	String	—	Name der Tabelle (nur sinnvoll bei Sekundärtabellen)
unique	boolean	false	Angabe, ob das Property ein eindeutiger Schlüssel ist
updatable	boolean	true	Angabe, ob die Spalte in Update-Anweisungen verwendet werden soll

1:1 Beziehung (unidirektional)

- Kaskadierendes Persistieren

```
@OneToOne(cascade = CascadeType.PERSIST)
private Adresse adresse;
```

```
Kunde kunde = new Kunde("Max", "Mustermann", ...);
Adresse adresse = new Adresse("Goethestrasse", ...);
kunde.setAdresse(adresse);
em.persist(kunde);
```

- Vollständige Kaskadierung

```
@OneToOne(cascade = CascadeType.ALL)
private Adresse adresse;
```

```
em.remove(kunde);
```

1:1 Beziehung (bidirektional)

- Mapping

```
@Entity
public class Kunde {

  @Id @GeneratedValue
  private Integer id;
  private String vorname;
  private String nachname;
  @Temporal(TemporalType.DATE)
  private Date geburtsdatum;
  private Anrede anrede;
  @OneToOne
  @JoinColumn(name = "adresse")
  private Adresse adresse;
  ...
```

```
@Entity
public class Adresse {

  @Id @GeneratedValue
  private Integer id;
  private String strasse;
  private String hausnummer;
  private String plz;
  private String ort;
  @OneToOne(mappedBy = "adresse")
  private Kunde kunde;
  ...
```

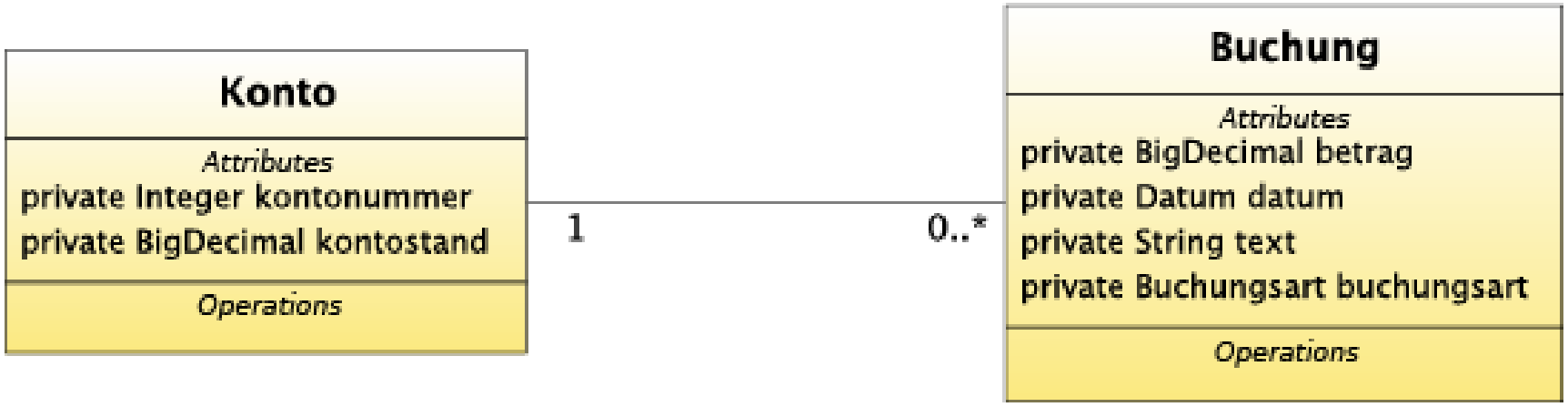
  - Achtung, der in der @JoinColumn Annotation verwendete Wert des Attributs name ist der Name der Tabellenspalte in der Datenbank,
  - während der in der @OneToOne-Annotation verwendete Wert des Attributes mappedBy der Name der Java-Property ist.

Abbildung von Objektbeziehungen

1:N UND N:1 - BEZIEHUNGEN

1:n und n:1 Beziehung

- Beispiel



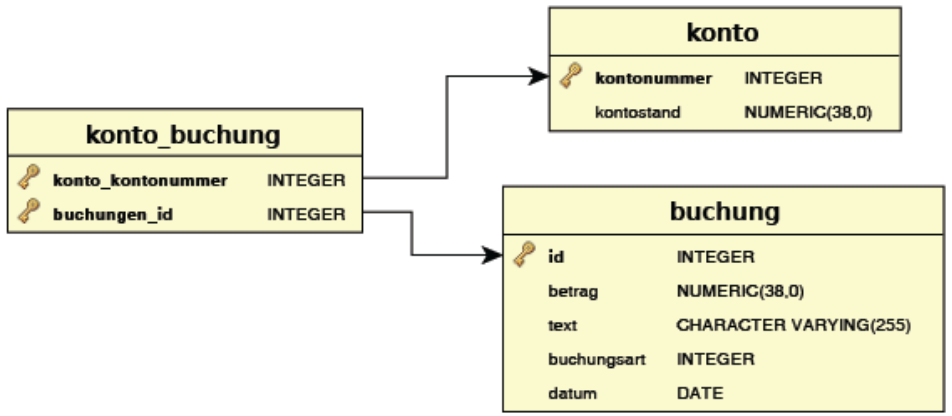
1:n und n:1 Beziehung (unidirektional)

- Mapping

```
@Entity
public class Konto {

  @Id @GeneratedValue
  private Integer kontonummer;
  private BigDecimal kontostand;

  @OneToMany(cascade =
             CascadeType.ALL)
  private Set<Buchung> buchungen;
  ...
```

```
@Entity
public class Buchung {

  @Id @GeneratedValue
  private Integer id;
  private BigDecimal betrag;
  @Temporal(TemporalType.DATE)
  private Date datum;
  private String text;
  private Buchungsart buchungsart;
  ...
```

- Datenbank



1:n und n:1 Beziehungen (unidirektional)

- Initialisierung

```
public Konto() {
  buchungen = new HashSet<Buchung>();
  kontostand = BigDecimal.ZERO.setScale(2);
}
```

- Befüllen und Persistieren

```
em.getTransaction().begin();
Konto konto = new Konto();
em.persist(konto);
Buchung buchung = new Buchung(new BigDecimal("600.00"), "12.08.2011",
                              "Miete", Buchungsart.SOLL);
konto.getBuchungen().add(buchung);
em.getTransaction().commit();
```

1:n und n:1 Beziehung (unidirektional)

- @OneToMany Attribute

@OneToMany(...)			
Attribut	**Typ**	**Default**	**Beschreibung**
cascade	CascadeType[]	—	Zu kaskadierende Operationen auf der Zielseite der Assoziation
fetch	FetchType	LAZY	Ladezeitpunkt der Assoziation (EAGER, LAZY)
mappedBy	String	—	Name des Property der Beziehung auf der Eigentümerseite. Bei bidirektionalen Beziehungen zwingend erforderlich
orphanRemoval	boolean	false	Angabe, ob Orphan gelöscht werden soll
targetEntity	Class	void.class	Zielklasse der Assoziation

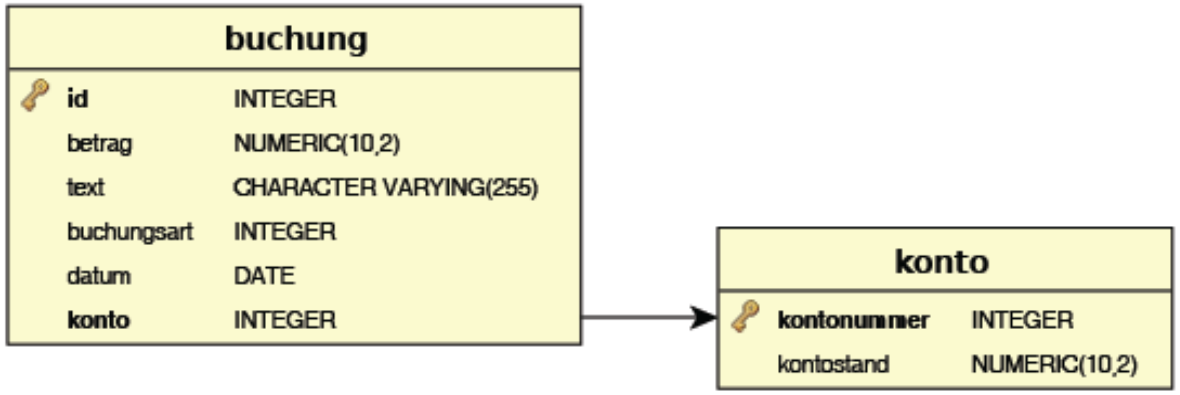
1:n und n:1 Beziehung (bidirektional)

- Mapping

```java
@Entity
public class Konto {

  @Id @GeneratedValue
  private Integer kontonummer;
  private BigDecimal kontostand;

  @OneToMany(mappedBy = "konto",...)
  private Set<Buchung> buchungen;
  ...
```

```java
@Entity
public class Buchung {

  @Id @GeneratedValue
  private Integer id;
  private BigDecimal betrag;
  ...
  @ManyToOne
  @JoinColumn(name = "konto",
              nullable = false)
  private Konto konto;
```

- Datenbank



1:n und n:1 Beziehung (bidirektional)

- @ManyToOne Attribute

@ManyToOne(...)			
Option	**Typ**	**Default**	**Beschreibung**
`cascade`	`Cascade-Type[]`	—	Zu kaskadierende Operationen auf der Zielseite der Assoziation
`fetch`	`Fetch-Type`	`EAGER`	Ladezeitpunkt der Assoziation (`EAGER`, `LAZY`)
`optional`	`boolean`	`true`	Beziehung ist optional
`targetEntity`	`Class`	`void.class`	Klasse des Ziels der Assoziation

1:n und n:1 Beziehung (bidirektional)

- Merkregel

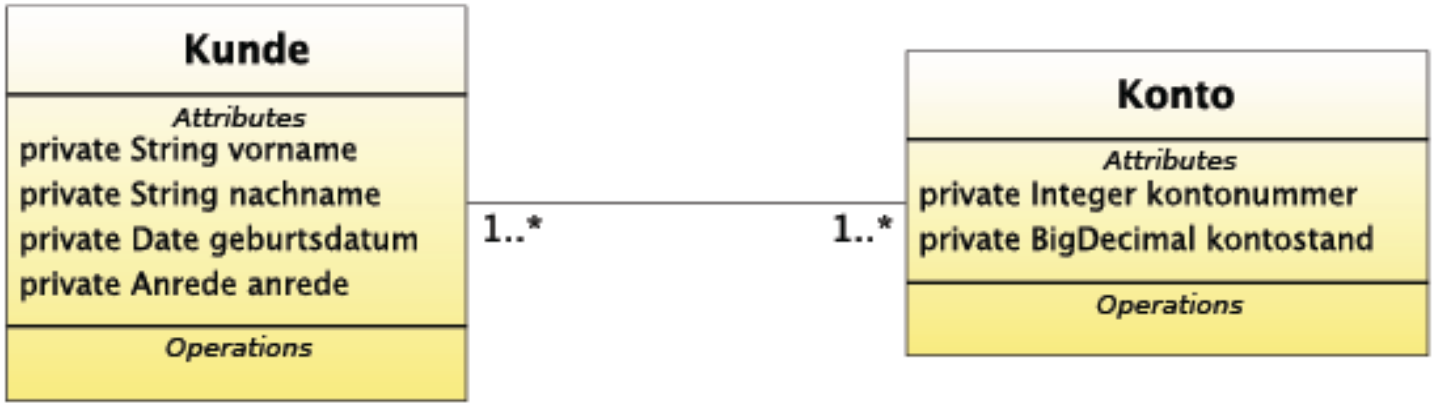
- Bei biderektionalen 1:n-Beziehungen (oder n:1) ist die „n-Seite“ die besitzende Seite.
- Die Join-Spalte der besitzenden Seite wird optional mit @JoinColumn annotiert.
- Die „1-Seite“ ist die inverse Seite, daher wird das mappedBy-Attribut verwendet.

Abbildung von Objektbeziehungen

N:M- BEZIEHUNGEN

n:m Beziehung

- Beispiel



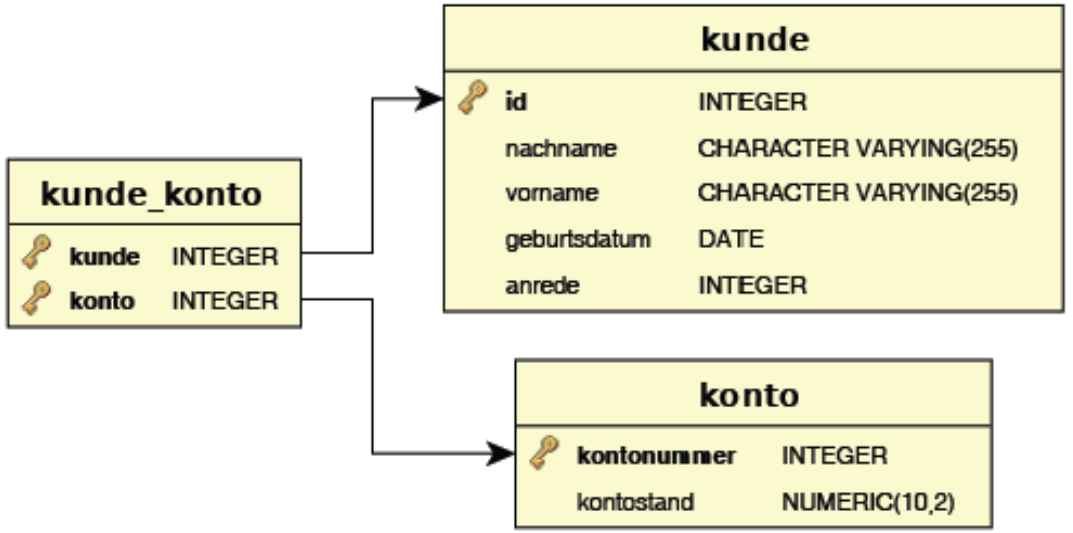
n:m Beziehung

- @ManyToMany Attribute

@ManyToMany(...)			
Option	**Typ**	**Default**	**Beschreibung**
`cascade`	`CascadeType[]`	—	Zu kaskadierende Operationen auf der Zielseite der Assoziation
`fetch`	`FetchType`	`LAZY`	Ladezeitpunkt der Assoziation (`EAGER`, `LAZY`)
`mappedBy*`	`String`	—	Name des Property der Beziehung auf der Eigentümerseite
`targetEntity`	`Class`	`void.class`	Zielklasse der Assoziation

n:m Beziehung

- Realisierung in Datenbank mittels JOIN Tabelle



n:m Beziehung

- @JoinTabelle Attribute

@JoinTable(...)			
Attribute	**Typ**	**Default**	**Beschreibung**
`catalog`	`String`	—	Katalogname
`inverse-JoinColumns`	`JoinColumn[]`	`{}`	Fremdschlüsselspalte(n) der Join-Tabelle zur nichtbesitzenden Tabelle
`joinColumns`	`JoinColumn[]`	`{}`	Fremdschlüsselspalten der Join-Tabelle zur besitzenden Tabelle
`name`	`String`	—	Name der Join-Tabelle
`schema`	`String`	—	Schema-Name
`unique-Constraints`	`unique-Constraint[]`	`{}`	Unique-Constraints (Benutzung nur bei Generierung)

n:m Beziehung

- Mapping

```
@Entity
public class Kunde {

  @Id @GeneratedValue
  private Integer id;
  ...
  @ManyToMany(cascade = PERSIST)
  @JoinTable(
    name = "Kunde_Konto",
    joinColumns =
      {@JoinColumn(name="kunde")},
    inverseJoinColumns =
      {@JoinColumn(name="konto")}
  )
  private Set<Konto> konten;
  ...
```

```
@Entity
public class Konto {

  @Id @GeneratedValue
  private Integer kontonummer;
  @Column(precision=10, scale=2)
  private BigDecimal kontostand;

  @ManyToMany(mappedBy ="konten")
  private Set<Kunde> kunden;
  ...
```

n:m Beziehung

- Codebeispiel

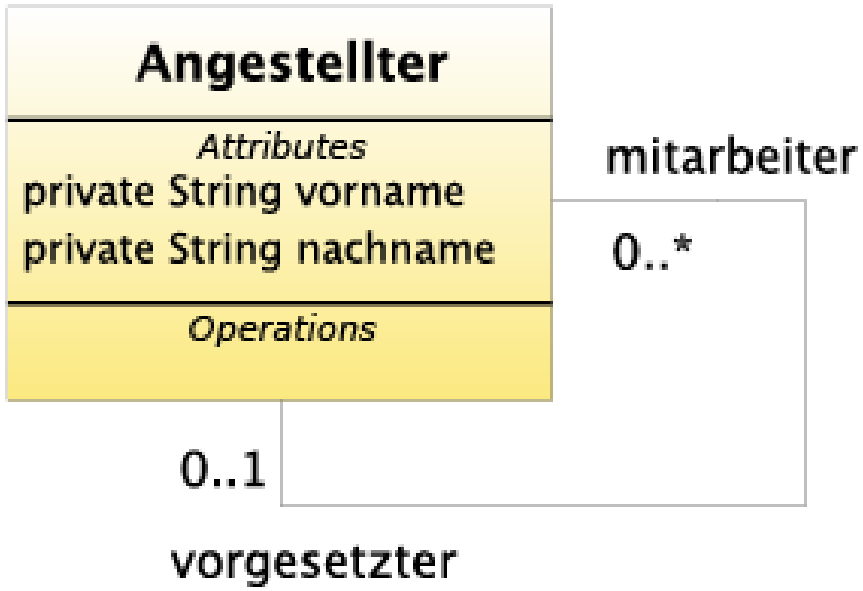
```
Kunde kunde = em.find(Kunde.class, ...);
for (Konto konto : kunde.getKonten()) {
  System.out.print("Das Konto " + konto.getKontonummer()
                   + " gehoert " + kunde.getVorname() + " "
                   + kunde.getNachname());
  for (Kunde k : konto.getKunden()) {
    if (!k.equals(kunde)) {
      System.out.print(" und " + k.getVorname() + " "
                       + k.getNachname());
    }
  }
  System.out.println();
}
```

Abbildung von Objektbeziehungen

REKURSIVE BEZIEHUNGEN

Rekursive Beziehung

- Beispiel

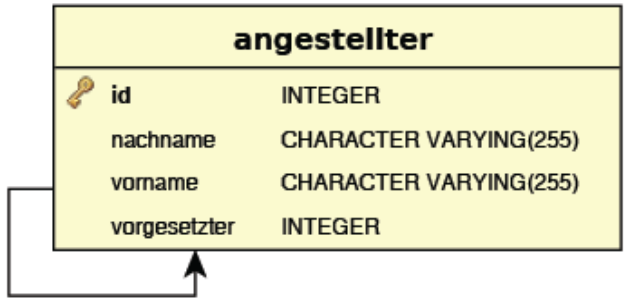


Rekursive Beziehung

- Mapping

```
@Entity
public class Angestellter {

  @Id @GeneratedValue
  private Integer id;
  private String vorname;
  private String nachname;

  @OneToMany(mappedBy = "vorgesetzter", cascade = CascadeType.ALL)
  private Set<Angestellter> mitarbeiter;

  @ManyToOne
  @JoinColumn(name = "vorgesetzter")
  private Angestellter vorgesetzter;
  ...
}
```

- Datenbank



Rekursive Beziehung

- Anlegen

```
public void neuerMitarbeiter(Angestellter angestellter) {
  this.getMitarbeiter().add(angestellter);
  angestellter.setVorgesetzter(this);
}
```

- Verwendung

```
public boolean isChef() {
  return vorgesetzter == null;
}

public int anzahlMitarbeiter() {
  return mitarbeiter.size();
}

public boolean isAbteilungsleiter() {
  return anzahlMitarbeiter() != 0;
}
```

Jakarta Persistence API

- **Transaktionen**

Objekte einlesen #1

- Der Objektzustand wird beim ersten Zugriff auf das Objekt eingelesen.
- Wenn **FetchType.EAGER** gesetzt ist, werden **referenzierte** Objekte ebenfalls **mitgeladen**.
- Wenn **FetchType.LAZY** gesetzt ist, werden referenzierte Objekte beim ersten Gebrauch eingelesen.
- Der Objektzustand wird nie automatisch aufgefrischt, nur via die **EntityManager.refresh()-**Methode.
- Eine neue Transaktion führt nicht automatisch zum erneuten Einlesen bestehender Objekte.

Objekte einlesen #2

- Der FetchType wird bei den Annotationen @Basic, @OneToOne, @oneToMany etc. als Attribut angegeben:

```
@Entity public class PMQ {
@ManyToMany( fetch = FetchType.EAGER )
protected List<Message> messages = new Vector<Message>();
@OneToOne( fetch = FetchType.EAGER )
protected Owner owner;
}


@Entity public class Message {
@ManyToMany( mappedBy = "messages", fetch = FetchType.EAGER )
protected List<PMQ> pmqs = new Vector<PMQ>();
}
```

- Defaultwerte für FetchType:
  - @OneToOne: EAGER
  - @OneToMany: LAZY
  - @ManyToOne: EAGER
  - @ManyToMany: LAZY

Objekte zurückschreiben

- Das Zurückschreiben findet zum **Commit-Zeitpunkt** oder explizit mit der **flush()** Methode statt.
- Das Zurückschreiben beinhaltet kein Refresh allfälliger Änderungen in der Datenbank in der Zwischenzeit.
- Das Zurückschreiben betrifft nur Änderungen, nicht ungeänderte Daten.
- Wenn der Persistence Context EXTENDED ist, bleibt ein Objekt im Zustand **managed** nach dem Commit.
  - Änderungen nach dem Commit werden aufbewahrt und im Rahmen der nächsten Transaktion in die Datenbank übernommen.
- Wenn der Persistence Context TRANSACTION ist, geht ein Objekt in den Zustand **detached** über nach dem Commit.
  - Änderungen müssen mit EntityManager.merge() innerhalb der nächsten Transaktion wieder eingekoppelt werden.

Isolationsgrad

- JPA arbeitet **standardmässig** im Isolationsgrad **READ COMMITTED**.
  - Gelesene Daten sind während der Transaktion nicht eingefroren.
  - Die Konsistenzprobleme **Lost Update** und **Phantom** können auftreten.

Sperrzeitpunkt

- Daten werden in der Datenbank im Rahmen von SQLBefehlen gesperrt.
- Da Änderungen vorerst lokal durchgeführt und erst zum **Commitzeitpunkt** in die DB propagiert werden, finden das **Sperren** erst beim **Commit** statt.
  - Allfällige Wartesituationen, Deadlocks, Integritätsverletzungen treten effektiv erst zum **Commitzeitpunkt** auf.
- Der Entity Manager stellt einen expliziten lock() Befehl zur Verfügung. (Das Verhalten ist stark implementationsabhängig).

Java Persistence API 2.1

- **Java Persistence Query Language (JPQL)**

Allgemeines zu JPA QL

- Auf das EJB-Modell ausgerichtete Abfragesprache zum Suchen und Bearbeiten von Entities, "SQL-Mapper".
- Abfragen operieren auf dem *Klassenmodell* aller Entitäten, nicht dem SQL-Datenmodell.
- Gegenüber EJB QL 2.1 wesentliche Erweiterungen:
  - direkte Update- und Delete-Operationen
  - OUTER |INNER JOIN, GROUP BY, HAVING Klauseln
  - Projektionen (Abfrage einzelner Attribute resp. Ausdrücke in der Select-Klausel)

EJB QL - Fremdschlüssel

- Im Objektmodell gibt es keine Fremdschlüssel (in der DB schon, aber diese sind in der Query Language nicht verwendbar). Es können also nur Felder/Beziehungsattribute verwendet werden, die im *Klassen*modell vorhanden sind. Will man beispielsweise alle Messages finden, die zu keiner PMQ gehören, so sind die Abfragen unterschiedlich, je nachdem ob zwischen PMQ und Message eine unidirektionale oder eine bidirektionale Beziehung definiert wurde (das relationale Modell ist für beide Fälle dasselbe).
- Abfrage bei **unidirektionaler** Beziehung von PMQ zu Message:
- Abfrage bei **bidirektionaler** Beziehung zwischen PMQ und Message:

```
select m
from Message m
where m not in ( select m2 from PMQ q JOIN q.messages m2 )
```

```
select m
from Message m
where m.pmq is null
```

Queries #1

1. Standardfunktion:

```
Query createQuery( String query )
```

Diese Funktion ist die häufigste Form. Sie übernimmt einen Query Text in der JPA Query Language (JPA QL).

2. NamedQuery

```
Query createNamedQuery( String qryname )
```

- Es kann im Java Source Code in Form einer Annotation ein Named Query definiert werden.
- Auf dieses Query kann mit createNamedQuery ("orphans") Bezug genommen werden. Anwendung: Wenn dasselbe Query häufig gebraucht wird.

```
@NamedQuery( name="orphans",
query=" select m from Message m where m.pmqs is empty " )
```

Queries #2

3. Direktes SQL

- Das Ausführen von direktem SQL, mit definiertem Mapping auf Felder einer Klasse

```
createNativeQuery( "select id, sender, receiver from Message",
                   "myMapping" )

 @SqlResultSetMapping(
 name="myMapping",
 entities={ @EntityResult(
       entityClass=pmq.Message.class,
       fields={
           @FieldResult( field="id", column="id" ),
           @FieldResult( field="sender", column="sender" ),
           @FieldResult( field="receiver", column="receiver" )
        }
 )}
)
```

Queries - Beispiele

- Unparametrisiertes Query

```
Query q = em.createQuery(
"select m from Message m where m.sender = 'Rolf'")
```

- Parametrisierte Queries

```
Query q = em.createQuery(
"select m from Message m where m.sender = :sender")

Query q = em.createQuery(
"select m from Message m where m.sender = ?1")
```

Queries - Parameterübergabe

- Übergabe eines Namensparameters

```
Query.setParameter( string name, Object value )
```

Beispiel

```
q.setParameter( "sender", "Rolf" );
```

- Übergabe eines Positionsparameters

```
Query.setParameter( int pos, Object value )
```

Beispiel

```
q.setParameter( 1, "Rolf" );
```

Queries - Auführen

- **Einschränken der Resultatmenge:**

```
setFirstResult( int pos );
setMaxResults( int max );
```

- **Abholen des Resultates**

```
List getResultList();
Object getSingleResult();
int executeUpdate();
```

- **Synchronisation**

```
setFlushModeType( FlushModeType type );
```

Queries - Beispiel

```
Query qry = em.createQuery( qrystring );
List result = qry.getResultList();
Iterator itm = result.iterator();
while( itm.hasNext() ) {
    Object o = itm.next();
    if ( o instanceof Object[] ) {
        for( Object x : (Object[])o )
        System.out.println( x.toString() );
    }
    else{
    System.out.println( o.toString() );
    }
}
```

Queries – Aufbau #1

```
select selectklausel
  from fromklausel
[ group by groupklausel ]
  [ having havingklausel]
  [ where whereklausel]
[ order by orderklausel ]
```

Queries - Aufbau #2

- In der **Selectklausel** kann eine Objektreferenz stehen (die in der Fromklausel definiert wird), ein Pfadausdruck, ein Aggregatausdruck oder ein Wertausdruck oder eine Kombination der drei letzteren.
- Die **Fromklausel** enthält die Grundmengen an Objekten mit einem Referenznamen (der in JPA QL obligatorisch ist) .
  Beispiel: `from PMQ q` .
  Werden mehrere Grundmengen angegeben ohne Join-Bedingung, wird das Kreuzprodukt gebildet wie in SQL.
- Die Groupklausel, die Havingklausel und die Orderklausel arbeiten sinngemäss **wie in SQL**.
  Insbesondere sind auch Pfadausdrücke erlaubt und die Orderklausel erlaubt die Zusätze ASC und DESC für die Sortierung.

Queries - Aufbau #3

- Zu erwähnen ist, dass keine der Klauseln Methodenaufrufe auf den Objekten erlaubt. Andere OO-Abfragensprachen, z.B. JDO Query Language erlauben dies.
- Die Whereklausel kennt die üblichen Vergleiche und Operatoren: =, >, >=, <, <=,<> (not equal), [NOT] BETWEEN, [NOT] LIKE, [NOT] IN, IS [NOT] NULL, AND, OR,NOT, (), [NOT] EXISTS.
- In der Whereklausel sind die speziellen Prädikate IS [NOT] EMPTY und [NOT] MEMBER OF zu erwähnen. Beide beziehen sich auf Pfadausdrücke, die in Collections enden.

Queries – Pfadausdrücke #1

- Ein Pfadausdruck ermöglicht die direkte Navigation von einem äußeren zu inneren, referenzierten Objekten:

```
select q.owner from PMQ q
select q.owner.name from PMQ q
```

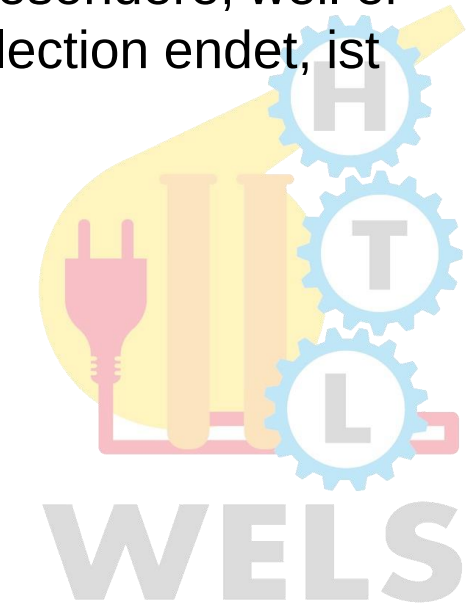
- der Ausdruck nimmt **keine Rücksicht** auf die **Kapselung** der inneren Objekte. Auch wenn das Ojekt Owner privat ist, kann darauf navigiert werden.
- Ist ein **Objekt** im Pfad **null**, so fällt die entsprechende Zeile aus dem Resultat weg.

Queries – Pfadausdrücke #2

- Ein Pfadausdruck kann in einer *Collection* enden:

```
select q.messages from PMQ q
```

- Ein Pfadausdruck, der in einer Collection endet *und* in einer select-Klausel steht, ist von der JPA Spez. eigentlich nicht erlaubt. Er soll aber hier als Illustration dienen, insbesondere, weil er beispielsweise in Hibernate zugelassen ist. Ein Pfadausdruck, der in einer Collection endet, ist vorallem für die Vewendung in der from- oder der where-Klausel vorgesehen.



Queries – Pfadausdrücke #3

- Ein Pfadausdruck kann nicht über eine Collection hinweg navigieren:

```
select q.messages.sender from PMQ q
```

- Der Versuch, solche Ausdrücke zu verwenden, ist ein häufiger Fehler. Bei genauerer Überlegung ist jedoch klar, dass das Konstrukt falsch ist. Wenn es zugelassen wäre, ist die Frage, ob sender ein Feld der Collection an sich, oder ein Feld der darin eingebetten Elemente ist. Es würde also eine Mehrdeutigkeit auftreten. Im Weiteren wäre zu bedenken, dass sender wieder eine Collection sein könnte. Damit wiederum kommen Probleme mit im Kreis herum referenzierten Objekten ins Spiel. Aus diesen Gründen wird auf dieses Konstrukt verzichtet in objektorientierten Abfragesprachen (Nicht hingegen im XML-Umfeld: XPath und XQuery bauen genau auf diesen Konstrukten auf).

JOIN-Abfagen

- Ein Join von zwei Grundmengen bedingt eine definierte Beziehung in den beiden Klassen:
- **Alle Messages der PMQ q1**

```
select m from PMQ q join q.messages m
    where q.qname = 'q1'
```

- **Alle PMQs mit Messages der Priorität 1**

```
select distinct q from PMQ q join q.messages m
    where m.priority = 1
```

- **Name jeder PMQ mit Absender aller Messages**

```
select q.qname, m.sender
    from PMQ q left join q.messages m
```

in und exists

Die Operatoren IN und EXISTS sind ähnlich wie in SQL, in der Unterabfrage können auch Pfadausdrücke verwendet werden:

- **Suche alle Queues mit Meldungen der Priorität 1**

```
select q from PMQ q
where exists (select m
                from q.messages m
                where m.priority = 1 )
```

- **Suche alle Messages, die zu einer PMQ von Rolf gehören**

```
select m from Message m
where m in ( select m
                from PMQ q JOIN q.messages
                where q.owner.name = 'Rolf' )
```

is empty, member of #1

- `is empty` und `member of` ergeben gut verständliche Abfragen auf Pfadausdrücken, die in Collections enden:

Suche leere PMQs

```
select q from PMQ q WHERE q.messages IS EMPTY
```

Alternative:

```
select q from PMQ q where not exists ( select m from q.messages )
```

is empty, member of #2

Suche nicht-leere PMQs

```
select q from PMQ q WHERE q.messages IS NOT EMPTY
```

Alternativen

```
select distinct q from PMQ q join q.messages m

select q from PMQ q where exists ( select m from q.messages )
```

is empty, member of #3

Suche PMQ's in denen die Message m vorkommt:

```
Message m = ... ; // irgendwoher
Query qry = em.createQuery(
      "select q from PMQ q where ?1 member of q.messages" );
qry.setParameter( 1, m );
List result = qry.getResultList();
```

Alternative:

```
select distinct q from PMQ q join q.messages m
            where m = ?1
```

Abfragen in Vererbungshierarchien

- Alle Objekte einer Klasse und ihrer Unterklassen:

```
select m from Message m
```

- Alle XMLMessages in einer PMQ:

```
select xm
from XmlMessage xm
where xm in ( select m
              from PMQ q JOIN q.messages m
              where q.qname = 'q1' )
```

- Alle XMLMessages in einer PMQ:

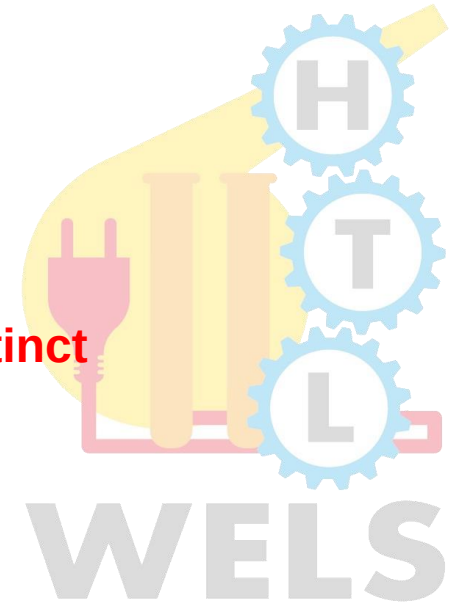
```
select m from Message m
where m in ( select mm from SimpleMessage mm )
```

Fetch Join

- Der **Fetch Join** erlaubt das Abfragen von Objekten, mit **gleichzeitigem Laden assoziierter Objekte**.
- Der Fetch Join ist Teil der **JPA Spez**.
- Der Fetch-Join ist ein **reines Optimierungs-Hilfsmittel**.
- Der rechte Operand des Fetch Join hat keine Identifikationsvariable und darf in der Abfrage nicht weiter verwendet werden.

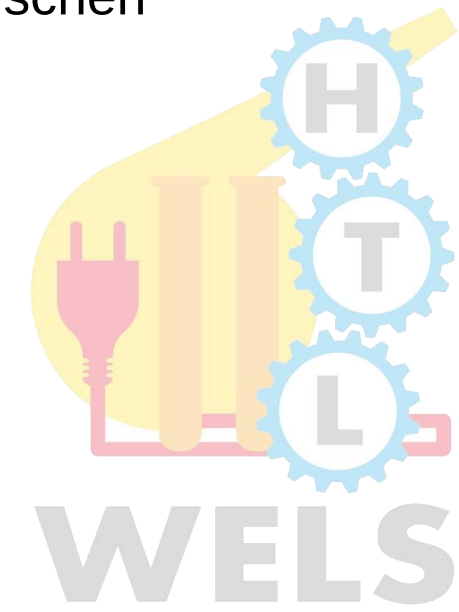
```
select q
from PMQ q left outer join fetch q.messages
where q.qname = 'q1,
```

- Der Fetch Join kann sowohl inner wie outer sein. Zu beachten: Die Join-Semantik hat natürlich zur Folge, dass die Abfrage eventuell mehrfach dasselbe Objekt zurückliefert. Ist dies nicht erwünscht, kann mit **select distinct** gearbeitet werden.



Update und Delete

- Gemäß Spezifikation können auch Update- und Delete- Operationen als "Abfragen" durchgeführt werden. Der Hintergrund dürften Performance-Überlegungen sein.
- Beispiel: `delete from PMQ q where q.qname = 'q1'`
- die Löschung beinhaltet kaskadierte Löschungen, die im Rahmen von Beziehungen definiert sind.
- Update- und Delete Befehle führen nicht automatisch eine Synchronisation zwischen Datenbank und Applikationscontext (Cache) aus.



JPA – Best Practices

- Siehe Dokument

Wichtige Links

- Generelle Info-Seiten von SUN/Oracle

  http://www.oracle.com/technetwork/java/javaee/overview/index.html

- Java Persistence API

  http://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html
  http://www.jcp.org/en/jsr/detail?id=317

- Einfaches Tutorial für JPA

  http://java.sun.com/developer/technicalArticles/J2SE/Desktop/persistenceapi/

- Liste aller Annotations

  http://www.oracle.com/technetwork/middleware/ias/toplink-jpa-annotations-096251.html

- Vergleich J2EE 2.1 und JEE 5.0/6.0

  http://www.oracle.com/technetwork/articles/javaee/jpa-137156.html